The Pragmatic Programmers

TURING 图灵程序设计丛书

The ThoughtWorks Anthology

软件开发沉思录

——ThoughtWorks文集

ThoughtWorks Inc. 著

ThoughtWorks中国公司 译

- 13篇美文汇聚软件开发精萃
- 来自软件界思想领袖们的经验心得
- 为你开启敏捷开发之门



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵程序设计丛书

软件开发沉思录

ThoughtWorks文集

The ThoughtWorks Anthology

Essays on Software Technology and Innovation

ThoughtWorks公司　著

ThoughtWorks中国公司　译

人 民 邮 电 出 版 社

北　　京

图书在版编目（CIP）数据

软件开发沉思录：ThoughtWorks 文集 / 美国 ThoughtWorks 公司著；ThoughtWorks 中国公司译 .—北京：人民邮电出版社，2009.9
（图灵程序设计丛书）
书名原文：The ThoughtWorks Anthology: Essays on Software Technology and Innovation
ISBN 978-7-115-21360-0

Ⅰ. 软…　Ⅱ. ①美…② T…　Ⅲ. 软件开发－文集　Ⅳ. TP311.52-53

中国版本图书馆CIP数据核字（2009）第155888号

内 容 提 要

本书对当前软件开发中存在的问题进行了广泛的探讨。包括公司创始人 Roy Singham 在内的许多 ThoughtWorks 员工参与到了这本文集的编写工作中，他们针对如何在软件开发生命周期中提高效率提出了多种可行性建议，内容涉及设计、架构、测试、领域特定语言的使用、构建和部署过程等。

本书的独特之处在于，它是由资深管理者和一线工程师共同创作完成的，各篇文章的作者以自己独到的视角对主题进行了分析，将其在特定领域积累的经验心得悉数奉上，从而能让更多的读者从中受益。本书条理清晰、思维严谨却又不乏生动活泼之处，而即便是书中专业性最强的文章，也不会让人觉得难以理解，除了技术人员外，本书对相关的非技术人员也很有价值。

图灵程序设计丛书

软件开发沉思录——ThoughtWorks文集

◆ 著　　　　ThoughtWorks公司
　译　　　　ThoughtWorks中国公司
　责任编辑　傅志红
　执行编辑　傅尔也

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：11.25
　字数：266千字　　　　　　2009年9月第1版
　印数：1－3 000册　　　　　2009年9月第1次印刷

著作权合同登记号　图字：01-2009-4817号

ISBN 978-7-115-21360-0

定价：39.00元

读者服务热线：(010)51095186　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

对本书的赞誉

从技术深度、详细程度、包含新观点/新成果的数量来看，书中文章各有千秋，但它们有一个共同的特点：密切关注实践。这群作者真正让本书既能启发思维，又能直接拿来参考，如此好书我已多年未见了。

——Stefan Tilkov

innoQ公司CEO

有这样一家企业，当整个IT行业都对定制软件开发的高难度与高成本望而却步时，他们却敢于挑战这个难题。这本文集让我们得以窥见这家企业内部百花齐放般的多样观点——这正是他们勇气和力量的源泉。

——W. James Fischer

埃森哲公司前CTO/资深合伙人

从CruiseControl等大获成功的开源项目，到各种技术博客和会议上鲜明的观点，你都能感受到ThoughtWorks公司对项目的影响力。身处其外的我们会不时想象这家公司内部究竟进行着怎样的讨论。这本书是一个难得的机会，让读者们可以深入幕后，参与到讨论之中——你会因此成为一个更出色的软件开发者。

——Nathaniel T. Schutta

资深技术专家，《Ajax基础教程》作者

软件开发很大程度上讲是一种团队活动，而团队的领导者则塑造了软件文化。成功的组织通常不会花时间来记录这些领导者的经验，于是其他人也无法学习这些经验并从中受益。这本有趣的文集由ThoughtWorks公司的一组领导者共同编撰，我们能透过这些文章瞥见ThoughtWorks公司的企业文化。

——Dave Thomas

Bedarra Research Labs

软件开发中最了不起的见解都出自那些为真实客户解决真实问题的人。然而除了在无数的博客中淘金之外，几乎没办法找到这些见解。十年来，ThoughtWorks人解决了大量真实问题，所以他们决定把自己的经验整理结集实在是一件令人欣喜的事。

——Gregor Hohpe

*Enterprise Integration Patterns*一书作者之一

这本精彩的文集论述了如何在当今的商业环境下恰当运用编程语言和工具来开发软件。这组作者都是软件世界里身经百战的老兵，他们的经验值得一看。

——Terence Parr

旧金山大学 ANTLR项目领导人

ThoughtWorks公司素来以其在软件开发上的经验与智慧闻名于世，这本文集出色地汇集了这些经验与智慧，使我们得以从中受益。这本书将作为一本重要参考书出现在每个项目组的书架上。

——Jeff Brown

G2One公司北美运营总监

序

本书面世之际，恰逢“敏捷中国2009大会”召开在即，两者可谓相得益彰。

从2004年进入中国，ThoughtWorks见证和参与了中国敏捷社区的发展历程：从五年前的筚路蓝缕，到如今的欣欣向荣。更令人欣慰的是，在原则、价值观等“大问题”上，敏捷的实践者们已经基本达成共识，社区的话题更趋于关注实践——这意味着敏捷社区正在步入成熟，社区成员将用他们的知识和技能为各自效力的企业创造更大的价值。

我们在这个时候翻译出版这本文集，是希望为社区的发展再尽绵薄之力。作为敏捷方法的积极推动者，ThoughtWorks从多年、多个行业的实践中积累了丰富的经验。本书收录的13篇文章涵盖了编程技术、项目管理、持续集成、测试等方面内容，将带领读者了解ThoughtWorks在软件生命周期各个环节所推荐的工作方式。

比较难得的是，这本文集不仅由ThoughtWorks员工撰写，也由ThoughtWorks员工翻译。译者们或是与文章作者素有私交，或是在文章所论述的领域有所专擅，这也使得翻译质量更有保障。感谢这些译者在工作之余的辛勤劳动，他们是韩锴、胡振波、金明、李剑、乔梁、熊节、徐昊、张晓庆和郑晔。

一本薄薄的文集当然不可能解决所有问题，我们更希望它能够起到抛砖引玉的作用。希望ThoughtWorks的经验心得能对国内的敏捷实践者们有所启发，帮助他们做出更多创新，创造更大价值。最后，希望你阅读愉快。

郭　晓

ThoughtWorks中国公司总经理

目　　录

第 1 章 引　　言

ThoughtWorks公司汇聚了一批热情洋溢、积极主动、才智过人的员工，他们为客户提供定制软件开发以及切合实际的咨询服务。如果你询问一位ThoughtWorks人，这家公司让他最喜欢的是什么，他很可能会告诉你：正是那些朝夕相处、并肩工作、彼此学习的同事们。这个群体里融合了技术极客、管理者、分析师、程序员、测试员和行政人员，他们来自不同的民族，有着不同的文化和教育背景。这种背景和视角的多样性，再加上坚持正确观点的热情，引发了很多活跃的讨论。

如今ThoughtWorks公司拥有近千名聪明而有见地的员工，在全球6个国家设有分支机构，组织内部几乎没有任何层级，并且一以贯之地坚持信息透明。可以说，我们已经创造了一家成功的企业。但我们对"成功"的定义远不止于此：一家企业的成功不仅意味着让客户满意，还应该对整个行业乃至整个社会产生积极的影响。我们有着更高的目标。

在博客世界里，在技术大会的会场中，在互联网上，在书架上，我们都能听到ThoughtWorks人的声音。在不断追求卓越的过程中，我们会近乎冷酷地剖析自己曾做过的事以及做这些事的方法，以寻求改进之道——在这方面我们永无满足。在上下求索之中，一旦学到了什么知识，我们就希望与他人共享。

我们曾在无数涉及各种领域、技术和平台的项目中磨砺自己的技艺。我们对软件开发有很多的思考，而这些思考全都植根于通过大量真实软件交付项目积累的经验中。工作性质的相对单纯也使得我们能够更加专注于软件开发这个领域。

没人会付钱给咨询师来闲聊公司的人力资源新规定，所以和大部分IT专业人士比起来，我们在工作时更加专注于软件交付。这使得我们更加注重实效，对细节把握得更加精确。

这本文集集中体现了ThoughtWorks人所面对的形形色色的IT问题。我们编纂这本文集不仅是为了展现"开发更好的软件"的几种方法，更是为了解决一个重要的问题：如何让IT投资转化为

实实在在的商业价值。在第一篇文章里，Roy开宗明义地指出，我们需要改变软件系统走完“最后一英里”投入实用的过程。他的主张直截了当：运营和部署问题同样应该成为开发过程中的核心问题，就跟需求分析和编程一样。我们必须记住，让代码通过QA部门的测试然后“一刀切”地丢给运营团队来处理部署、运营等麻烦事，这是无法确保项目成功的。开发软件的团队必须知道，只要软件还未最终投入使用，他们的工作就不算完成。然而Roy的文章可不止是耍耍小聪明、重新定义一下“完成”和“成功”之类的词汇。他希望读者认真思考何时、如何让项目各方参与到敏捷过程中来。我们一直用自己的聪明才智来制造和改进编码阶段的工具（例如自动构建工具、支持脚本编程的测试工具、重构工具等），同样的聪明才智也能够用于解决“最后一英里”问题。

在这本文集中，你会看到Roy的号召不断得到回应。例如James谈到了性能测试：这件事经常被习惯性地忽视，直到项目晚期才被想起，这时很多设计决策已经在代码中根深蒂固，为了解决性能问题想要撤销这些设计决策改走它途，而又不破坏好容易才正常工作的业务功能，不啻于尝试颠覆热力学第二定律。James提出了一条切实可行的路子：他不仅指出应该从项目之初就定义性能需求（谁还能置疑这一点呢？），而且对“如何获取有用的性能需求”这一问题进行了探讨。他不是一味高喊“尽早测试”，而是实际探讨在哪里、如何运行这些测试。

Julian的文章主题是“Ant脚本重构”。他列举了大量标准的重构手法，针对每个重构手法又给出了清晰的范例。对于任何需要应对不断增长的构建脚本的人，这篇文章都是一个绝佳的参考。Dave的文章介绍了关于“一键部署”的概念框架，恰好与Roy的开篇文章交相辉映。他在文中探讨了一些大问题，例如管理程序生成的巨大的二进制文件、与部署环境中的异构系统集成等。有助于提高软件开发效率的各种技巧最终都会被用于软件部署，Dave的文章朝着这个方向又迈进了一步。

Stelio的文章介绍了如何运用沟通技巧来获悉项目的健康情况。他给出了一些度量标准——既有客观标准，也有主观标准——并探讨了如何有效地展现度量结果，以使所有相关人员每天都能在同一个“仪表板”前工作。他致力于把项目的关键指标展示给尽可能多的相关人员。而这又引出了另一个概念：项目人类学。Tiffany的文章读起来就好像玛格丽特·米德[①]关于萨摩亚的研究报告。她介绍了一种全新的项目成员角色——迭代经理，并指出这个角色应该如何融入群体。她敏锐地察觉到了对团队组织方式加以微调从而提升其效率的可能性，迭代经理这个角色正可以促成这样的变化。Jeff的“九诫”则让我想起大师们对编程之道追随者的训导。这些“诫条”简洁明了，真要身体力行却绝非易事（尤其是它们要求程序员“抛弃”如此之多的习惯）。

① 玛格丽特·米德（Margaret Mead，1901—1978），美国人类学家，美国现代人类学成形过程中最重要的学者之一。她在1928年出版了《萨摩亚人的成年人》一书，探讨了正值青春期的萨摩亚少女的性与家庭风俗，针砭美国社会对待青少年的方式，轰动一时。——编者注

Rebecca的文章在我看来颇有些润物细无声的味道。在进入“语言战争”之前，她首先带领读者对各种语言进行分类。阅读这篇文章，就像是与林奈[1]在花园中漫步：起初你们看到各种植物独有的特征，然后将特征泛化抽象成为一个框架，以此对看到的植物分门别类。Rebecca打下了一个扎实的基础，却将出人意表的观点放在最后：这不止是一篇关于时下流行的编程语言的调查报告，更揭示了软件开发的多样性——至于“Java vs. .NET”这样的语言之争只不过是一系列源远流长的讨论的最新版本而已。真正重要的是了解自己想要解决的问题，以及用哪种工具能最好地解决这些问题。此时的Rebecca仿佛进入了一间凌乱的工作室，她先把扳手和锤子分别收好，然后在工具箱上贴好标签，让后来人一看便知其中的工具应该用来干什么。

其他的文章更加关注技术，却也更能展现我们这些并肩工作的同事们多样的才华。Ian提出了一个综合方案：把SOA契约考虑成消费者（而非顾客）驱动的。他的文章更尝试解决一个由来已久的问题：如何构建和发展共享服务，使之随时应对业务需求的变化，而又不影响现有消费者继续使用服务。Erik也在思考类似的问题：在一个设计良好的系统中，领域模型与基础设施会应当解耦，但这就要求基础设施层使用领域模型所展现的元数据。从“用什么类来表现某个领域元素”这样的设计决策中，我们可以读出一些隐含的元数据，但这样的信息远不足以实现更多的操作，例如有效性验证。一些流行的编程语言（例如Java和C#）以标注（annotation）和特性（attribute）的形式提供了更丰富的元数据表现形式，Erik的文章则以案例分析的方式展现了如何充分利用这些特性。Martin为守护巢穴的狂徒们设计了几种DSL，这让我想起了多年前学用C编程的情形：当时我的参考书是Kernighan和Ritchie的经典教材，我震惊于他们对字符串复制函数使用了一些迭代，就使其变得极为简洁而优雅，达到了我自己随后的编程努力所无法企及的高度。

这本文集所收录的文章虽然各自成篇，彼此之间却有着千丝万缕的联系。它们就像同一片布满迷雾的IT丛林中的林间小径，它们或显而易见、或出人意表。文章选题涉猎之广，解决问题的办法差异之大，恰能反映众位作者所在的这个组织为各种思想的萌生营造了一个健康的环境。阅读这本文集恰如管中窥豹，让我迫不及待地想看到这些才华横溢的同事们为整个行业、整个社会创造更多。

Mike Aguilar
ThoughtWorks公司副总裁
2008年2月15日

① 林奈（Linnaeus，1707—1778），瑞典植物学家、冒险家，他首先构想出定义生物属种的原则，并创造出统一的生物命名系统。——编者注

第2章

走完业务软件的“最后一英里”

Roy Singham，公司创始人、董事长
Michael Robinson，中国公司技术总监

测试驱动设计（TDD）、持续集成（CI）、结对编程、重构等敏捷实践让我们能够快速交付高质量的软件。采用敏捷方法开发的软件从项目初期就能可靠地工作，并且在之后随需应变的过程中也能持续可靠工作。

但挑战仍然存在，尤其是在软件开发的“最后一英里”——所谓“最后一英里”，是指软件满足了功能需求之后，尚未投入实际运行并创造业务价值的阶段。

软件开发者——尤其是面对交付压力的软件开发者——常常对“最后一英里”视而不见。但它确实正在成为业务软件交付中最大的压力点。

2.1 “最后一英里”问题的源头

“最后一英里”问题往往是日积月累而来的。可以想象你白手起家创立了一家企业，除了极具创意的商业模式之外你什么都没有：没有既有系统，没有历史交易数据，没有客户信息，连收入都没有。这时你需要的只是一个简单的系统来验证业务模式的可行性。如果你的点子行得通，你会追加投资来改进系统的功能和可伸缩性。但眼下你的资金有限，所以你得尽快让系统上线，看到效果。

在这种情况下，系统能够很快部署到生产环境。这时不存在“最后一英里”问题：只要满足业务需求，软件几乎立即就能投入使用。

时间流逝，一穷二白的小公司变成了赚钱的成功企业，现在你有了很多客户以及两年来的交易数据。你还购买了一套客户关系管理（CRM）系统和一套财务软件，并且将它们与核心业务系统集成起来。

原来的核心业务系统也增加了一些新功能。这时，一个全新的商业机会出现了，你需要开发另一套核心业务系统来抓住这个机会，而且还得将它与现有系统整合。

这时事情就开始变得棘手了。在让第二套核心业务系统上线之前，必须进行足够的测试，以保证它能够与遗留系统和数据可靠交互。

又是几年时间过去，这家小公司已经成长为一家跨国上市企业，拥有许多条业务线和庞大的市场份额。公司有上万名员工，客户更是数以百万计。公司的营收额巨大而且仍在增长，投资者和市场分析师们每天紧盯着相关数据。最初的核心业务系统如今已经容纳了8年来的历史交易数据，并且与12个关键系统集成。系统早已补丁摞补丁、扩展再扩展，但不断增加的交易量和不断变化的业务需求还是令它不堪重负。公司希望重新开发一套系统来取代这套老系统，从而充分利用最先进的技术。

这就是存在“最后一英里”问题的软件。

2.2 理解问题

从商业的角度来说，之所以要开发一套新的软件，是因为它能创造更多业务价值。但如果下列情况之一发生的话，开发新软件反而会得不偿失：

- 新的软件无法负载业务模式所需的用户量或交易量；
- 新的软件破坏了遗留数据库中的数据；
- 新的软件出现无法预期的错误，或是导致遗留系统变得不可靠；
- 新的软件把敏感数据暴露给了不可信的用户；
- 新的软件使恶意用户能够进行未经授权的操作。

新软件给现有业务带来的风险很可能超过它所能提供的价值，所以很自然地，公司越大，在引入新软件时就必须越发小心。这也就导致了旧系统和过时的技术堆积在产品环境中，而不是被替换掉。而与这些堆积的遗留系统集成这一难题又会增加引入新软件的成本与风险。于是这就成了一个恶性循环。

这个恶性循环使得企业越来越难以改变业务模式，因为做出改变的成本与风险越来越高。没有技术包袱的新公司或许可以在一段时期内将老迈的对手甩在身后，但最终它们自己也会走上同样的衰老之路。

敏捷软件开发能够为快速接纳不断变化的需求提供保证。对于那些被信息技术、无所不在的互联网和全球化趋势驱赶着的企业而言，这种对响应能力的保证如今更具吸引力了：他们需要加

快商业模式创新的步伐。

但从业务负责人的角度来看，软件开发是一件“端到端”的事情——他们批准预算，然后他们希望看到软件实际投入应用。至于中间发生了什么，他们兴趣不大。所以对于业务负责人来说，只有当软件能够更快投入使用时，所谓“敏捷过程”才真正是敏捷的。

如果业务软件本身很庞大，又要整合复杂的遗留系统，那么新版本发布时很可能需要用上三四个月甚至更长的时间来安装、测试并逐渐稳定，唯有如此才能确保新软件的安全。在这样的环境里，敏捷开发者们或许用一两周时间就能实现用户提出的新需求，但很可能还得再过半年才能让用户真正用上这些新功能——这取决于发布周期的长短，以及如何对新功能进行确认。

或许这也可以被看作一个“敏捷大获成功”的故事——两周就完成任务！至于那六个月的延期嘛，那是别人的事……但是，这种看待问题的方式是错误的。

2.3　解决“最后一英里”问题

如今的敏捷软件开发项目常需要经历以下步骤：

(1) 业务主管定义需求；
(2) 业务主管找老板批准预算；
(3) 开发团队定义一堆故事；
(4) 开发团队把故事完成；
(5) 业务主管接受做好的故事；
(6) 开发团队移交代码。

如果第(3)、(4)、(5)步都顺利走过，团队能按时甚至提前到达第(6)步，这就是个成功的项目。项目的成果将是一个通过了所有验收测试的软件，然后这个软件就被交付，如果运气好的话几个月以后业务主管将会开始使用它。

如果运气不那么好，软件会被客户退回来修改。优秀的开发者们会想办法防止这种情况发生。他们会精心设计测试，会结对编程，诸如此类。而且很多时候这些办法确实奏效。但即便如此，走完“最后一英里”的过程还是严重拖延了软件创造业务价值的时间，哪怕交付的代码本身一点问题也没有，哪怕项目比计划提前交付。

除非将敏捷软件开发变成一个端到端的软件交付过程，否则“最后一英里”就是个无解的难题。“如何部署到生产环境”的问题必须在每个环节中得到关注。从这个角度出发，我们将重新

审视软件开发中人与自动化工具所扮演的角色。

2.4　人

敏捷思潮的一大贡献是对于软件开发作为社会行为的认识：有更好的沟通，才会有更好的软件。在采用敏捷开发实践时，很大部分的努力都是用来打破现有组织架构的藩篱，代之以更高效的模式与实践。

然而迄今为止，敏捷实践者们关注的焦点几乎都是软件开发者与使用者之间的沟通。他们之间的沟通改善了需求的质量，使大家对业务目标达成共识，但非功能性需求该怎么办呢？谁对这些需求负责？如何在沟通中体现它们？这些问题往往得不到回答。

要避免这种“代码一刀切”的软件开发方式，最简单的方式是找出对非功能的、跨模块的需求负责的人，让他们也参与到软件开发的“社交活动”中来。尽早、尽可能频繁地让他们参与沟通。同样，这也可能需要打破现有组织架构的藩篱，代之以更高效的模式与实践。

举例来说，软件编写好之后，系统管理员要负责安装和配置。他们还要监控系统在生产环境下的运行情况，确保操作正确无误。如果有问题出现，他们要按照预先制定的流程来恢复系统状态。他们需要规划系统安装之初和规模扩展之后的硬件需求——内存、磁盘、网络、电源、冷却，凡此种种。

维护和支持人员需要系统提供有用的错误报告和有效的诊断流程。他们需要知道如何帮助用户解决简单的系统故障，以及在遇到严重的系统故障时如何上报。

很多行业还有相关的监管人员需要其他方面的信息，例如确保系统实现了法律规定的隐私保护或者数据安全要求。他们要求系统符合必要的审计需求。

这些负责人各自的业务需求都切实合理，必须得到满足。在开发过程中越早考虑到这些需求，软件就可能越快投入使用。让相关负责人更早地参与沟通，他们的需求就可能更快、更有效地得到实现。

2.5　自动化

目前很多软件项目的“最后一英里”发布过程是手工完成的。这样的过程低效而易错，因此耗时甚多、代价不菲。为了大幅减少耗费在“最后一英里”的时间，必须尽量将一切可能自动化的工作自动化。这需要改变我们构造软件的方式。

一旦开始将开发过程自动化，开发团队很快就会发现：自动化与软件开发有着很多相似的问题。构建脚本需要测试和调试，这些测试也会失败，也需要时时更新。但由于这些自动化工具经常被认为“不是软件”，开发团队经常无意解决其中存在的问题，尽管这方面早有大量成熟经验。

构建脚本、测试脚本、安装脚本、配置文件，它们都是端到端交付物中的一部分，都对最终的生产系统做出贡献，所以应该将它们与代码一视同仁。在“开发交付物”和“生产交付物”之间不应该有任何差异对待：两者都应该被纳入项目的版本控制仓库；两者都应该有清晰的组织和一致的维护；并且，对两者都应该进行重构，以简化其结构、复用其功能。最后，所有的系统组件——操作系统、应用服务器、数据库、防火墙、存储系统等——都应该支持有效的自动化测试。整体系统架构在设计和定义时就应该充分考虑自动化测试。

对于存在大量遗留系统的环境而言，要在集成点上完全支持自动化测试或许不切实际。在这种情况下，对集成点做模仿（mock）至少聊胜于无。但所有新系统在暴露集成点时都应该同时提供自动化测试机制（例如设置测试环境、回滚测试环境、记录测试结果、读取测试结果等）。

在新软件发布之前对其进行验证的一个好办法是“回放”从当前系统采集到的一段时间内的事务记录，并将结果与当前系统进行比较。新系统应当支持自动化的“回放”测试。

2.6　针对自动测试非功能性需求的设计

非功能性需求（Nonfunctional requirements，NFR）可能不是系统功能细则的一部分，但无论如何它们也是合理的需求，而且往往有着重要的业务价值。“最后一英里”的测试工作中很大部分都是在确认系统是否满足了所有非功能性需求，例如响应时间、事务吞吐量、可访问性、安全性等。但这些NFR测试经常是在软件“完工”以后才开始的。

NFR测试应该在编码之前就开始。尤其是性能和资源占用方面的需求，应该提前加以分析，并用一个模型将功能代码与非功能需求关联起来。

一种传统的性能测试方法大致是这样：在项目开始时就指定“某操作的系统响应时间不超过5秒”。然后开发者编写软件。当他们开发完成以后，管理员把软件安装在预上线系统上，测试人员来执行该操作，一边看着手表，默数系统到底用了多少秒。

这种做法有两大问题。首先，如果该操作花了5分钟而不是5秒钟，相关的软件恐怕得全部重写，而这时人们正在等待将其投入使用。其次，即便开发团队为了避免类似问题而编写了自动化的性能测试，并在持续集成环境中运行这些测试，测试环境与生产环境的巨大差异也经常使测试结果变得毫无意义，而保持这两个环境的一致性往往并不可行。

然而，换一个角度，我们可以说“生产环境可以负载每秒500次随机磁盘存取，因此系统必须在2 500次随机磁盘存取之内完成该操作”。大部分操作系统环境都提供了详细的性能计数器，可以很容易地将其与自动化测试结合。与基于时间的测试相比，基于计数器的测试对环境的依赖更小。能执行测试的地方越多，测试就会被执行得越频繁。此外，基于计数器的性能测试粒度也细得多，于是开发团队将更有可能在编写软件的同时发现性能或者资源方面的问题——此时修复这些问题是最容易、最省事的。

性能与资源占用的测试模型同样是端到端交付的一部分，它应该被纳入版本控制，应该得到持续的维护，并且能够在不同环境下运行。用于收集和查看计数器数据的库和工具应该让开发者也学会使用，这样他们就能更快捷有效地把代码与自动化的性能测试结合起来。如果测试模型做得足够好，最后的预发布测试就应该能够很快完成，并且不会带来什么出人意料的坏消息。

2.7 将设计与生产环境分离

持续集成与测试驱动设计已经成为快速开发高质量软件的利器。快速的反馈让开发团队能够在错误出现之初就迅速地修复，从而在工作中保持这样一种信心：系统的功能始终是符合预期的。

可惜的是，“部署到生产环境”这一过程迄今为止从CI和TDD中获益甚少。理想情况下，从用户和运维的角度针对完全集成的生产环境编写测试应该不太困难，CI系统也应该能够以足够快的速度运行这样一套完备的测试，从而随时验证软件的行为。但实践中通常会有一些阻碍导致这美好的理想难以成真。

首先，生产环境往往庞大、复杂、昂贵并且难以搭建。其次，针对这种环境的端到端的测试往往难以设计、难以编写、也难以验证。第三，即便搭建了适当的环境，也写好了测试，这样的测试运行起来通常很慢，运行这样一套完备的测试可能需要花上几天时间。

解决最后一个问题的办法是并行地执行测试：如果能够同时执行10个测试，那么整套测试执行的速度通常就可以提高10倍。但第一个问题——庞大、复杂、昂贵——又使得并行测试多少有些不切实际：搭建并维护10套生产环境，只为开发团队执行测试之用，对于绝大多数项目来说都是无法想象的奢侈。

但如果能够降低生产环境的成本，如果能够很快地搭建生产环境，如果能让多个开发团队分担这些成本，那么并行执行大量测试就变得比较现实了。虚拟化技术的最新发展正好提供了这样的可能性。借助最新的虚拟化技术，我们可以快速地保存和还原整个虚拟的生产环境，然后在大量廉价硬件上复制它。

当然，如果所要开发的系统在虚拟测试环境与真实生产环境的行为有所不同，那这种做法就大错特错了。这时自动化测试的结果没有什么可信度，而且对开发或测试的效率也无所助益。

每个业务软件都会对部署环境有所依赖和假设。这些假设和依赖既可能是显性的，也可能是隐性的。如果一个系统对部署环境存在大量隐性的假设和依赖，想要在另一个环境中编写出有意义的端到端系统测试就会相当困难。

所以，为了能够在不同环境下进行快速、完善的系统测试，系统的设计者必须识别出隐性的假设和依赖，并使它们变成显性的、可测试的假设和依赖。同时设计者还必须系统地减少对环境的假设与依赖。

一旦有可能执行一套完备的自动化系统测试，编写这些测试的投入就会显得更划算。归根到底，我们希望CI系统能够给开发团队和运维团队提供这样的信心：正在开发的软件是随时都可以部署到生产环境的。

2.8　无版本软件

敏捷过程的价值，就在于减少从“提出业务需求”直到“软件投入使用来满足业务需求”这两个端点之间所需的时间与成本。将这一目标推向极致，就可以想象出这样的情形：当客户提出一个功能需求，一旦开发完成，这个新的功能就可以直接投入使用。

如今，一些简单的小型网站已经采用了这样的开发模式。有时新功能从开始开发到投入使用用不了一个小时。这时开发者们也不再需要“发布版本号”这样的东西，他们做好一个功能就可以发布一个功能。

然而对于庞大、复杂而又敏感的遗留环境来说，“无版本软件”顶多可以作为一个美好的远景目标。“最后一英里”的发布工作如此繁琐，我们不得不把开发好的功能打成大包，每隔一段相当长的时间才发布一次。这种情况在可预见的将来也不大会有所改变。

尽管如此，现今通行的做法毕竟代价高昂。人力的浪费已无须赘述，错失商业机会所造成的间接损失更是难以衡量。在软件开发的总成本中，这部分成本正在占据越来越大的比重。

虽然或许无法达到“无版本软件”的境界，但我们至少可以有所进步。很多改进措施是立竿见影的，而且我们还能找到更多改进措施，只要开动脑筋去主动寻找。“最后一英里”问题不会一夜之间消失无踪，但只要保持对它的关注，让各方人员积极参与端到端的软件开发过程，问题总会得到解决。

第3章

一个巢穴，二十种Ruby DSL

Martin Fowler，首席科学家

Ruby在最近得以流行的主要原因是它非常适合用来编写内部领域特定语言（Internal DSL）。内部领域特定语言是指在另一种语言（宿主语言）之上编写的领域特定语言。目前，用Ruby编写DSL更有日趋火爆的迹象。

内部领域特定语言是一个在Lisp圈子里一直非常流行的想法。很多Lisp语言拥护者炮轰Ruby在这方面没有带来任何新意。但令Ruby与众不同的一点是：它提供了多种技术来开发内部DSL。虽然Lisp也提供了一些很好的机制，但相对于Ruby，它的选择还是较少的。

本章通过一个例子来探讨这些技术，以使你对这些技术有所体会，并让你能在它们之间进行比较取舍。

3.1 巢穴

接下来，我将用一个简单的例子来探讨这些技术。这个例子是一个常见但很有趣的抽象配置问题。在各种各样的设备中，我们都会遇到这样的问题：如果要有x，那么必须要有一个相匹配的y。当我们购买电脑、安装软件或做其他一些更加有趣的事情时，都会遇到这个问题。

在这个例子中，让我们想象有这么一家公司，它专门将复杂的设备提供给那些脑子里整天想着要征服世界的狂徒。从此类电影的数量来看，这是一个很大的市场。而这些狂徒藏身的巢穴，被一些极具魅力秘密特工不断捣毁，更增加了对这些设备的持续需求。

本章将用DSL来描述这些狂徒们放置在巢穴中设备的配置规则。此例中的DSL将会描述两类事物：物件（item）和资源（resource）。物件表示一些具体的事物，比如摄像机（camera）和酸性溶液（acid bath）等；而资源表示一些大量的原材料，比如电（electricity）等。

此例涉及两种资源：电和酸（acid）。假设每种资源都拥有多种不同的属性。比如，所有物件的电力都由巢穴中的电厂供应（这些狂徒们可不想使用社会公共服务）。所以在这个抽象表示中，每件资源都需要有它自己独立的类。

为了更好地描述这个问题，让我们假设只有两种类别的资源：简单的和复杂的。简单资源（比如电）只有为数不多且数量固定的属性，所以可以在生成函数的参数中传入这些属性。而复杂资源（比如酸）有很多可选的属性，它们需要一些单独的设置方法来设置属性。虽然此例中的酸实际上只有两种属性，但不妨让我们把它想象成拥有几十种不同属性的复杂资源。

而关于物件，有三个特点需要声明：它们使用资源，它们提供资源，并且它们依赖于巢穴中的其他物件。

好了，不继续吊你的胃口了，让我们马上来看看这个抽象表示的实现吧。注意，在所有将要讨论的例子中，都将使用同一个抽象表示。

下载 lairs/model.rb

```
class Item
  attr_reader :id, :uses, :provisions, :dependencies
  def initialize id
    @id = id
    @uses = []
    @provisions = []
    @dependencies = []
  end
  def add_usage anItem
    @uses << anItem
  end

  def add_provision anItem
    @provisions << anItem
  end
  def add_dependency anItem
    @dependencies << anItem
  end
end

class Acid
  attr_accessor :type, :grade
end

class Electricity
  def initialize power
    @power = power
  end
  attr_reader :power
end
```

把所有的特定配置放在一个配置对象中。

下载 lairs/model.rb

```
class Configuration
  def initialize
    @items = {}
  end
  def add_item arg
    @items[arg.id] = arg
  end
  def [] arg
    return @items[arg]
  end
  def items
    @items.values
  end
end
```

为了描述清楚本章所要阐述的问题，我们只需定义少量物件以及它们的规则：

- 一种使用12单位电和五级盐酸的酸性溶液；
- 一个使用1单位电的摄像机；
- 一个供应11单位电并且依赖于一个安全排气口（secure air vent）的小型电厂（small power plant）。

在抽象表示中可以这样描述这些物件的规则。

下载 lairs/rules0.rb

```
config = Configuration.new
config.add_item(Item.new(:secure_air_vent))

config.add_item(Item.new(:acid_bath))
config[:acid_bath].add_usage(Electricity.new(12))
acid = Acid.new
config[:acid_bath].add_usage(acid)
acid.type = :hcl
acid.grade = 5

config.add_item(Item.new(:camera))
config[:camera].add_usage(Electricity.new(1))

config.add_item(Item.new(:small_power_plant))
config[:small_power_plant].add_provision(Electricity.new(11))
config[:small_power_plant].add_dependency(config[:secure_air_vent])
```

虽然能形成可用的配置但这样的代码并不流畅。下面，我们将尝试用不同的方法来编写代码，以更好地描述这些规则。

3.2 使用全局函数

函数是程序最基本的结构，它是生成软件和为程序引入领域名称的最简单方式。

所以，我们编写DSL的初次尝试就是调用一系列的全局函数。

下载 lairs/rules8.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(acid)
acid_type(:hcl)
acid_grade(5)
uses(electricity(12))

item(:camera)
uses(electricity(1))

item(:small_power_plant)
provides(electricity(11))
depends(:secure_air_vent)
```

这些函数名组成了DSL的词汇表：`item`声明了一个物件，`uses`表明一个物件使用了一种资源。

这段DSL描述的配置规则其实就是在建立各个对象之间的关系。当描述一个摄像机使用1单位电时，就是在建立这个物件和电这种资源之间的关联。在第一个巢穴表示中，关联关系是由命令的顺序决定的。`uses(electricity(1))`这个命令紧跟着摄像机的声明，所以它被应用于摄像机这个物件。我们可以说，关联关系是由语句的顺序上下文隐式定义的。

计算机和人不同，我们可以通过阅读DSL文本而获知命令的顺序，但计算机需要一些额外的帮助才能理解它们。当计算机导入DSL时，可以用一些特殊的变量来跟踪上下文关系，这些变量被称作上下文变量。用一个上下文变量来跟踪当前使用物件的代码如下。

下载 lairs/builder8.rb

```
def item name
  $current_item = Item.new(name)
  $config.add_item $current_item
end

def uses resource
  $current_item.add_usage(resource)
end
```

因为使用了全局函数，所以需要使用全局变量来作为上下文变量。这么做确实不十分恰当，但我们马上会看到在很多语言中都可以避免全局函数的使用。事实上，使用全局函数只是权宜之计。

我们也可以采用同样的方法来处理酸（acid）的属性。

下载 lairs/builder8.rb

```
def acid
  $current_acid = Acid.new
end
def acid_type type
  $current_acid.type = type
end
```


用顺序关系来描述物件和资源之间的关联还凑合能用，但用它来描述具有依赖关系的物件之间的关联时就显得不太合适。这时候需要在它们之间建立一些显式的关联。比如，可以在声明一个物件时（item(:secure_air_vent)）赋给它一个标识符，在以后需要使用它时用那个标识符引用物件（depends(:secure_air_vent)）。当然，上面描述的小型电厂依赖于安全排气口的关联是通过顺序关系建立的。

物件和资源的不同之处在于，资源就是Evans所说的值对象（value object）[Eva03]，它们只被物件关联。而物件可以在DSL中通过依赖关系被任意关联。所以，物件需要一些标识符以备在将来引用。

Ruby用symbol来处理这类标识符：:secure_air_vent。symbol是以冒号开头的、不含空格的字符串。很多主流语言都没有这种数据类型。在这种特殊的用途中，可以把它们当作字符串一样看待。但symbol并不具备很多字符串操作，而且所有等值的symbol均共享一个实例，这使得对symbol的搜索更加高效。但在此例中使用它们的主要原因，是symbol所表达的含义正好跟我在这里使用它们的意图不谋而合，即把它们当作符号使用。我把:secure_aire_vent作为一个符号，而不是一个字符串。可以看到，选择一种正确的数据类型可以帮助我们更加清晰地表达我们的意图。

另一种方式当然是使用变量。但在DSL中我不太喜欢使用变量。变量的问题就在于，它们是可变的。它们可以被赋予不同对象，因此我们不得不跟踪变量中的对象到底是什么。变量是一种有用的工具，但对它们的跟踪非常棘手。在DSL中我们通常应该避免使用它们。而标识符始终指向同一个对象，不会改变。

对于物件依赖关系的表述，标识符是必须的，我们同样可以用它代替顺序关系来描述资源。

下载 lairs/rules7.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(:acid_bath, acid(:acid_bath_acid))
acid_type(:acid_bath_acid, :hcl)
acid_grade(:acid_bath_acid, 5)
uses(:acid_bath, electricity(12))

item(:camera)
uses(:camera, electricity(1))
```

```
item(:small_power_plant)
provides(:small_power_plant, electricity(11))
depends(:small_power_plant, :secure_air_vent)
```

标识符的使用意味着关联关系的显式定义，而且意味着全局上下文变量的多余。这是一举两得的事情：我喜欢清晰地定义关系，而且我也讨厌使用全局变量。但这样做的代价是DSL看起来会更加冗长，值得用一些隐式机制使DSL变得更加清晰易读。

3.3　使用对象

如前所述那样使用函数的一个主要问题是：需要使用全局函数。过多的全局函数会导致对它们的管理非常困难。使用对象的一个好处是，可以把函数归于类中。通过合理地安排DSL代码把函数从全局作用域中移出，并且放置在更加合理的地方。

类方法和方法链

在面向对象语言中控制方法作用域的最简单方式是使用类方法。但类方法却带来了大量重复：在每次调用类方法时都需要使用类名。我们可以通过方法链来减少重复，就如下面的代码一样。

下载 lairs/rules11.rb

```
Configuration.item(:secure_air_vent)

Configuration.item(:acid_bath).
    uses(Resources.acid.
      set_type(:hcl).
      set_grade(5)).
    uses(Resources.electricity(12))

Configuration.item(:camera).uses(Resources.electricity(1))

Configuration.item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```

每个DSL子句都从一个类方法的调用开始。类方法返回一个对象，即下一个方法调用的接收者。通过这样不断地返回下一个调用的接收者把所有的方法调用串起来。但在有些地方使用方法链却不太合适，这时就应该重新使用类方法。

让我们更进一步地看一下这个例子，以了解它的具体实现。不过此例中有一些问题和错误，我会在以后探讨和更正它们。首先从一个物件的定义开始吧。

下载 lairs/builder11.rb

```
def self.item arg
  new_item = Item.new(arg)
  @@current.add_item new_item
  return new_item
end
```

这个方法创建了一个新物件，然后把它置于一个存储配置信息的类变量中，最后返回这个新创建的物件。最后的物件返回是这里的关键，因为它建立起了方法链。

下载 lairs/builder11.rb

```
def provides arg
  add_provision arg
  return self
end
```

这个provides方法只是调用了一下add方法，然后立即返回它本身以让方法链得以延续。其他方法也大多类似。

方法链的使用与很多好的编程准则是相矛盾的。很多语言都约定修饰符（改变对象状态的方法）不得返回任何东西，这遵循了命令-查询分离原则（command query separation principle）——一个在大多数时候都值得遵守的原则。但不幸的是它跟流式的内部DSL相悖。DSL编写者为了支持方法链经常会抛弃这个准则。同时，此例也使用了方法链来设置酸的类型（type）和级别（grade）。

另一个与常规编码准则大有区别的地方是代码格式。在这个例子中，代码被刻意编排成层级关系突出以适应DSL的需求。在使用方法链时，你会经常看到方法调用会换行。

除了演示如何编写方法链，此例还展示了如何使用工厂类来创建资源。我们并不是往Electricity类中添加创建资源的方法，而是定义一个资源类，资源类包含创建电和酸实例的类方法。这种工厂经常被称为类工厂或者静态工厂，因为它们只包含用于创建对应对象的类（静态）方法。类工厂使得DSL更加易读，而且避免了需要在领域类中添加一些额外方法的问题。

这更加突出了这段DSL代码的一个问题：为了让这段代码工作，我们需要在领域类中添加许多方法——而这些方法放置在领域类中并不合适。一个对象中的大多数方法都应该在独立的调用中具有意义，但DSL方法的编写是为了让方法在DSL表达式中更有意义。所以，DSL代码和普通代码需要遵循不同的原则，比如命名和命令-查询分离等原则。此外，DSL方法跟上下文密切相关，并且只能用于DSL表达式中来创建对象。基本上，编写DSL方法需要遵循的原则和普通方法是不一样的。

表达式生成器

避免DSL和常规API冲突的一个方法是使用表达式生成器模式（Expression Builder pattern）。这个模式的本质是把DSL方法放置在用于创建真实领域对象的一个独立对象中。使用表达式生成器模式的方式有两种。其中一种方式是保持DSL代码不变，但用DSL方法创建生成器对象而非领域对象。

可以通过让原来的类方法返回一个不同的物件生成器对象来实现这种方式。

下载 lairs/builder12.rb

```
def self.item arg
  new_item = ItemBuilder.new(arg)
  @@current.add_item new_item.subject
  return new_item
end
```

物件生成器提供了DSL代码所需的方法，然后把这些方法的调用转发给物件对象。

下载 lairs/builder12.rb

```
attr_reader :subject
def initialize arg
  @subject = Item.new arg
end
def provides arg
  subject.add_provision arg.subject
  return self
end
```

当然，编写代码时我们可以完全摆脱领域对象的API，让DSL看起来更加清晰。

下载 lairs/rules14.rb

```
ConfigurationBuilder.
  item(:secure_air_vent).
  item(:acid_bath).
    uses(Resources.acid.
       type(:hcl).
       grade(5)).
    uses(Resources.electricity(12)).
  item(:camera).uses(Resources.electricity(1)).
  item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```

上面的代码从一个生成器的使用开始，然后使用生成器本身的方法链。这不仅避免了重复，而且避免了讨厌的类变量的使用。第一个调用是调用配置生成器的类方法来创建一个配置生成器实例：

下载 lairs/builder14.rb

```
def self.item arg
  builder = ConfigurationBuilder.new
  builder.item arg
end
def initialize
  @subject = Configuration.new
end
def item arg
  result = ItemBuilder.new self, arg
  @subject.add_item result.subject
  return result
end
```


创建一个配置生成器，紧接着即调用新创建物件的实例方法。这个实例方法创建了一个新的物件生成器，并且把它返回以作进一步的调用。在这里稍微有点怪异的是一个类方法和一个实例方法使用了相同的名字。为了避免引起混乱，通常我不会这样做。但我又一次打破了惯例，因为它会让DSL看起来更加流畅。此物件生成器具有和之前一样的获取物件信息的方法。另外，它需要一个物件方法来定义一个新物件。

下载 lairs/builder14.rb

```
def item arg
  @parent.item arg
end
def initialize parent, arg
  @parent = parent
  @subject = Item.new arg
end
```

为了在需要时重新回到配置生成器，在创建物件生成器时传入配置生成器作为它的父生成器。同时这也是为了让物件生成器在记录依赖时能查找到其他物件。

下载 lairs/builder14.rb

```
def depends_on arg
  subject.add_dependency(configuration[arg])
  return self
end
def configuration
  return @parent.subject
end
```

而之前我需要从全局变量或者类变量中查找其他物件。

最后一个优化是重命名酸生成器的方法，以让它们更加易读。生成器的存在使得我们不需要担心底层领域对象的命名冲突。

为每一个领域对象创建一个生成器并不是表达式生成器唯一的使用方法。另一种思路是为所

有领域对象创建一个生成器对象。以下就是为上面同样的DSL编写的新生成器。

下载 lairs/builder13.rb

```
def self.item arg
  result = self.new
  result.item arg
  return result
end
def initialize
  @subject = Configuration.new
end
def item arg
  @current_item = Item.new(arg)
  @subject.add_item @current_item
  return self
end
```

这段代码不再每次都创建一个新对象，而是用一个上下文变量来跟踪当前使用的物件。这也意味着我们不再需要定义父生成器的向前跟踪方法。

更多方法链

方法链是一个好工具，但是否可以在所有地方都使用它呢？我们是否可以去掉资源工厂呢？事实上的确可以，去掉之后DSL代码就会变成下面这样。

下载 lairs/rules2.rb

```
ConfigurationBuilder.
  item(:secure_air_vent).

  item(:acid_bath).
    uses.acid.
       type(:hcl).
       grade(5).
    uses.electricity(12).

  item(:camera).uses.electricity(1).

  item(:small_power_plant).
    provides.electricity(11).
    depends_on(:secure_air_vent)
```

（请注意我在这里添加了一些空行来提高Ruby代码的可读性。）

使用方法链还是参数是一个时常碰到的问题。当参数是直接量的时候，比如`grade(5)`，那么使用方法链就过于复杂。在复杂和简单之间，我倾向于后者，这是个显而易见的选择。难以选择的时候是在碰到如`uses.electricity...`和`uses(Resources.electricity...`这样的代码时。

随着方法链的增多，生成器的复杂度也随之增加。一个很好的例子是：在引入附属对象后生成器的复杂性急速增加。资源在两种情况中使用，跟随着uses或者是跟随着provides。因此，如果要使用方法链，就需要跟踪资源使用的环境，才能正确地响应对electricity的调用。

另一方面，使用参数会使我们失去方法链所带来的对作用域的控制，所以在参数创建时需要提供作用域控制——此例使用的是工厂提供的类方法。同时，引用工厂名也是我乐意去避免的不断重复的麻烦事之一。

引入参数引起的另外一个问题是：DSL编写者经常需要在使用何种方法之间进行抉择，这会使DSL的编写变得困难。

由于在这方面经验不够，我也无法给你提供一个明确的建议。当然我觉得首选是使用方法链，因为作为一种技术它拥有很多的支持。但在使用时需要注意使用方法链所带来的复杂性。一旦生成器的实现开始变得混乱（我知道这是一个含糊的说辞），就使用参数。随后我就会介绍一些技术来避免引入参数时带来的类工厂重复问题，但那取决于我们使用的是何种宿主语言。

3.4　使用闭包

闭包是语言中一个越来越常见的特性，特别是在一些支持内部DSL的动态语言中。闭包给一个等级式结构引入了一个新的作用域上下文，这对于DSL特别适合。下面就是使用了闭包的“巢穴”程序示例。

下载 lairs/rules3.rb

```
ConfigurationBuilder.start do |config|
  config.item :secure_air_vent

  config.item(:acid_bath) do |item|
    item.uses(Resources.acid) do |acid|
      acid.type = :hcl
      acid.grade = 5
    end
    item.uses(Resources.electricity(12))
  end

  config.item(:camera) do |item|
    item.uses(Resources.electricity(1))
  end

  config.item(:small_power_plant) do |item|
    item.provides(Resources.electricity(11))
    item.depends_on(:secure_air_vent)
  end
end
```

此例放弃了方法链的使用，而且每个方法都有一个清晰的接收者。接收者通过宿主语言的闭包语法来设定。DSL中往往会有等级式结构，这使方法调用的嵌套变得更加简单。

DSL代码需要的嵌套布局正好由宿主语言的嵌套结构所提供，这使代码的布置更加简单。另外，变量（比如item和acid）的作用域也被限制在宿主语言的块中。

显式方法接收者的使用代表方法链的多余。这也意味着当领域对象本身的API已经具有这个功能时，生成器也是多余的。在这里，我们仍对item使用生成器，但对acid使用真实的领域对象。

使用这项技术的一个限制是它需要宿主语言提供对闭包功能的支持。虽然也可以使用临时变量模拟，但临时变量带来的问题是它们不能很好地控制作用域，除非你提供一个额外的作用域机制。无论有没有额外的作用域控制，代码都不会如DSL那么流畅，而且容易出错。闭包通过绑定作用域和定义变量很好地避免了这个问题。

3.5 执行上下文

到目前为止，我们还没有谈到DSL代码的执行上下文。如果没有定义执行上下文，谈论一个没有接收者的函数调用和数据项就完全没有意义。之前我们假设执行上下文是全局的，比如函数foo()就被假设为一个全局函数。我们已经谈到过如何使用方法链和类方法在其他作用域中调用函数，但我们仍然能够改变整个DSL程序的上下文。

提供执行上下文最简单的方法是把DSL代码嵌入一个类中，这样DSL代码就可以使用类的其他方法和字段（field）。在支持开放类的语言中，可以通过直接打开一个类来实现；而在其他语言中，需要定义一个子类来实现。

下面就是一个定义子类的例子。

下载 lairs/rules17.rb

```
class PrimaryConfigurationRules < ConfigurationBuilder
  def run
    item(:secure_air_vent)

    item(:acid_bath).
      uses(acid.
         type(:hcl).
         grade(5)).
      uses(electricity(12))

    item(:camera).uses(electricity(1))

    item(:small_power_plant).
```

```
      provides(electricity(11)).
      depends_on(:secure_air_vent)
  end
end
```

把DSL代码放置到一个子类中，允许我们做一些在全局执行上下文中无法做到的事情。比如我们可以抛弃方法链来实现对item的连续调用，因为我们可以让item成为配置生成器中的一个方法。同样，我们可以把acid和electricity定义成配置生成器中的方法，以此来避免静态工厂类的使用。

但这么做的缺点是DSL文本上将会出现一些额外的类、方法头和尾。

此例演示了如何在一个对象实例的上下文中执行DSL代码。这非常有用，因为对象实例的上下文允许我们访问实例中的变量。我们也可以通过类方法在一个类上下文中这么做。通常我更喜欢使用实例上下文，因为它允许我们创建一个生成器实例，并且在执行完DSL代码后就可抛弃这个实例。这样可以保证两个执行环境的相互隔离，以避免残留数据相互干扰的风险（特别是在需要处理并发时）。

而Ruby提供了一个两全其美的方法：Ruby中有个方法叫做instance_eval，它可以接收一段代码——一个字符串或者是一个块，然后在一个对象上下文中执行这段代码。这使得我们只需要在文件中保存DSL代码，而仍能把代码置于一个对象上下文中执行。

下载 lairs/rules1.rb

```
item :secure_air_vent

item(:acid_bath).
  uses(acid.
       type(:hcl).
       grade(5)).
  uses(electricity(12))

item(:camera).uses(electricity(1))

item(:small_power_plant).
  provides(electricity(11)).
  depends_on(:secure_air_vent)
```

Ruby在支持闭包的同时也支持执行上下文的更改，这使得我们可以把拥有闭包的代码传给instance_eval，然后在一个对象实例中执行。用这种方式写就的代码如下。

下载 lairs/rules18.rb

```
item :secure_air_vent

item(:acid_bath) do
  uses(acid) do
```

```
    type :hcl
    grade 5
  end
  uses(electricity(12))
end

item(:camera) do
  uses(electricity(1))
end

item(:small_power_plant) do
  provides(electricity(11))
  depends_on(:secure_air_vent)
end
```

结果是很具吸引力的。这段代码具有闭包结构，且作为显式接收者的块参数没有重复。然而这种技术的使用需要格外小心，因为块上下文的切换容易引起很多混乱。在每个块中伪变量self指向不同的对象，这会迷惑DSL编写者，特别是需要从块中获取标准的self时。

这种混乱在实际应用中已被证实。Ruby的生成器库在早期时使用了instance_eval，但实践中却发现它会引起混乱并且难以使用。Jim Weirich（Ruby生成器库的作者）总结道：如果DSL编写者是程序员，像这样切换执行上下文对他们而言是个坏消息，因为它违背了我们对宿主语言的期望（这个担心引起了其他Ruby DSL编写者的共鸣）。而对于非程序员的DSL编写者来讲这不是一个很大的问题，因为他们本来就没有这种期待。我个人的感觉是：内部DSL跟宿主语言的集成度越高，就越应该避免这种违背正常期望的行为。而对于一些没有必要跟宿主语言很相似的迷你语言（mini-languages），就比如本章中的这个配置例子，应该以易读性为重。

3.6　字面量集合

对内部DSL而言，函数调用语法是一个很重要的结构机制。事实上对很多语言而言，函数调用基本上是唯一的结构机制。而在有些语言中有另一个很有用的机制：在表达式中使用字面量集合。但在很多语言中这个机制受到局限，或者是因为没有简单的语法，或者是因为不能在合适的地方使用它们。

有两种非常有用的字面量集合：列表和映射（也可以叫散列、字典和关联数组）。大多数现代语言都在库中提供了这样的对象，以及用来处理这些对象的合适的API。用这两个结构编写DSL都非常方便，虽然有些Lisp程序员会告诉你可以通过列表来模拟映射。

下面就是一个用字面量集合来定义acid的例子。

下载 lairs/rules20.rb

```
item :secure_air_vent
```

```
item(:acid_bath) do
  uses(acid(:type => :hcl, :grade => 5))
  uses(electricity(12))
end

item(:camera) do
  uses(electricity(1))
end
item(:small_power_plant) do
  provides(electricity(11))
  depends_on(:secure_air_vent)
end
```

上述代码混合使用了函数调用和字面量集合，并且利用了Ruby可以在没有二义性时清除参数括弧的特性。acid的函数现在看起来是这样的。

下载 lairs/builder20.rb

```
def acid args
  result = Acid.new
  result.grade = args[:grade]
  result.type = args[:type]
  return result
end
```

用一个字面量散列作为参数是Ruby一项惯例（这是它受Perl的影响之一）。这种方式对于有些函数非常合适，比如有很多可选参数的创建方法。应用于这个例子，这种方式不仅提供了一个干净清晰的DSL语法，而且避免了acid和electricity生成器的使用——取而代之以直接创建需要的对象。

如果更进一步利用字面量集合，会出现什么情况？比如当我们把对uses、privides和depends_on这些函数的调用都替换成映射时。

下载 lairs/rules4.rb

```
item :secure_air_vent

item :acid_bath,
  :uses => [acid(:type => :hcl,
                 :grade => 5) ,
            electricity(12)]

item :camera,
  :uses => electricity(1)

item :small_power_plant,
  :provides => electricity(11),
  :depends_on => :secure_air_vent
```

使用这种方法有利有弊。对于像小型电厂这种简单物件它非常合适。但对于如酸性溶液这种复杂物件，它却不合适。酸性溶液依赖于两种资源，所以需要把对acid和electricity的调用放

在一个列表中。一旦把它们置于字面量映射中，代码就变得很不直观。

接下来实现会变得更加复杂。对item方法的调用既需要名称，也需要映射。在Ruby中会将其视作一个name参数后面跟着一个“名称–值”对形式的多重参数。

下载 lairs/builder4.rb

```
def item name, *args
  newItem = Item.new name
  process_item_args(newItem, args) unless args.empty?
  @config.add_item newItem
  return self
end
```

process_item_args函数根据不同的键来切换处理每个闭包，要注意的是：args中的值可能是一个元素，也可能是一个列表。

下载 lairs/builder4.rb

```
def process_item_args anItem, args
  args[0].each_pair do |key, value|
    case key
    when :depends_on
      oneOrMany(value) {|i| anItem.add_dependency(@config[i])}
    when :uses
      oneOrMany(value) {|r| anItem.add_usage r}
    when :provides
      oneOrMany(value) {|i| anItem.add_provision i}
    end
  end
end
def oneOrMany(obj, &block)
  if obj.kind_of? Array
    obj.each(&block)
  else
    yield obj
  end
end
```

当你遇到这种情况——传入的参数值既可能是一个单独的元素，也可能是一个列表——时，始终把参数作为列表传入通常会让情况变得比较简单。

下载 lairs/rules21.rb

```
item :secure_air_vent

item :acid_bath,
  [:uses,
    acid(:type => :hcl, :grade => 5),
    electricity(12)]

item :camera,
  [:uses, electricity(1)]
```

```
item :small_power_plant,
  [:provides, electricity(11)],
  [:depends_on,  :secure_air_vent]
```

上面代码中item方法的参数是一个名称和一个列表（而不是散列）。列表中的第一个元素是键，其后的元素是这个键对应的值（这就是Lisp程序员用列表模拟散列的方法）。这种方法减少了嵌套层次，并且更易处理。

下载 lairs/builder21.rb

```
def item name, *args
  newItem = Item.new name
  process_item_args(newItem, args) unless args.empty?
  @config.add_item newItem
  return self
end
def process_item_args anItem, args
  args.each do |e|
    case e.head
    when :depends_on
      e.tail.each {|i| anItem.add_dependency(@config[i])}
    when :uses
      e.tail.each {|r| anItem.add_usage r}
    when :provides
      e.tail.each {|i| anItem.add_provision i}
    end
  end
end
```

在这里需要注意的是，我们把列表当作了一个头和尾的组合（而不是一系列元素）来处理。所以不要用只有两个元素的列表来替换散列，因为那没有任何价值。在这里我们用第一个元素为键，其他元素为值的列表替换散列，这样我们就不需要在一个集合中嵌套另一个集合。

头和尾并非是Ruby的列表（叫做`Array`）默认具有的方法，但添加它们非常简单。

下载 lairs/builder21.rb

```
class Array
  def tail
    self[1..-1]
  end
  alias head first
end
```

在结束字面量集合的讨论之前，让我们来看一看最后版本。下面就是以使用映射为主，列表为辅的整个配置代码。

下载 lairs/rules22.rb

```
{:items => [
  {:id => :secure_air_vent},
  {:id => :acid_bath,
    :uses => [
```

```
      [:acid, {:type => :hcl, :grade => 5}],
      [:electricity, 12]]},
  {:id => :camera,
    :uses => [:electricity, 1]},
  {:id => :small_power_plant,
    :provides => [:electricity, 11],
    :depends_on => :secure_air_vent}
  ]}
```

下面是只使用列表实现的版本，也叫Greenspun版本[①]。

下载 lairs/rules6.rb

```
[
  [:item, :secure_air_vent],

  [:item, :acid_bath,
    [:uses,
        [:acid,
            [:type, :hcl],
            [:grade, 5]],
        [:electricity, 12]]],

  [:item, :camera,
    [:uses, [:electricity, 1]]],

  [:item, :small_power_plant,
    [:provides, [:electricity, 11]],
    [:depends_on, :secure_air_vent]]]
```

可变参数方法

有些语言支持可变参数方法，此时在函数调用中使用字面量列表是一种很有用的技术。在下面的代码中，我把这种方式应用于uses方法。

下载 lairs/rules24.rb

```
item :secure_air_vent

item(:acid_bath) do
  uses(acid(:type => :hcl, :grade => 5),
       electricity(12))
end

item(:camera) do
  uses(electricity(1))
end

item(:small_power_plant) do
  provides(electricity(11))
```

① 出自Philip Greenspun的“第十编程法则”：任何使用静态类型检查语言编写的、足够复杂的程序都包含一个特定、非正式定义、容易引入Bug且缓慢的动态检查语言实现。——译者注

```
  provides(electricity(11))
  depends_on(:secure_air_vent)
end
```

在这种需要把方法调用中的列表组合在一起的情况下，使用可变参数是非常趁手的——尤其是当语言对在何处放置字面量列表限制得非常严格时。

3.7　动态接收

动态编程语言的一个特性是：它能对没有在接收对象里定义的方法调用进行响应。

让我们在这个例子中探索一下这句话的含义。到目前为止，我们假设巢穴里的资源数量是相对固定的，我们可以编写相应的代码来处理这些固定数量的资源。但如果不是假设的这种情况呢？如果有很多资源呢？如果需要把非常多的资源置于配置中呢？

下载 lairs/rules23.rb

```
resource :electricity, :power
resource :acid, :type, :grade

item :secure_air_vent

item(:acid_bath).
  uses(acid(:type => :hcl, :grade => 5)).
  uses(electricity(:power => 12))

item(:camera).
  uses(electricity(:power => 1))

item(:small_power_plant).
  provides(electricity(:power => 11)).
  depends_on(:secure_air_vent)
```

`electricity`和`acid`仍旧是生成器中的方法。我希望这些方法可以创建新定义的资源，但我不希望去定义这些方法，而是希望它们能够根据资源的数据来自动创建。

在Ruby中可以通过重写`method_missing`方法来实现。在Ruby中，如果一个对象接收了一个没有定义的方法调用，那么它就会执行`method_missing`方法。这个方法默认是从`Object`类中继承而来，而且会抛出一个异常。我们可以通过重写这个方法来做一些有趣的事情。

首先做好对资源方法调用的准备工作。

下载 lairs/builder23.rb

```
def resource name, *attributes
  attributes << :name
  new_resource = Struct.new(*attributes)
  @configuration.add_resource_type name, new_resource
end
```

Ruby有一个用来创建匿名类的工具，叫做*struct*。当需要一个资源时，调用struct进行定义。以resource方法的第一个参数作为新定义资源的名字，并根据随后的参数设置新定义资源的属性（property）。然后把这些新定义的资源保存于配置中。

接着我重写了method_missing方法。该方法在一个字面量字典中遍历了所有新定义的资源，以确定方法名是否与新的struct之一对应。如果有，则加载这个struct。

下载 lairs/builder23.rb

```
def method_missing sym, *args
  super sym, *args unless @configuration.resource_names.include? sym
  obj = @configuration.resource_type(sym).new
  obj[:name] = sym
  args[0].each_pair do |key, value|
    obj[key] = value
  end
  return obj
end
```

在method_missing被调用时，首先确认是否有资源可以响应这个调用。如果没有，则调用超类中的method_missing以引发一个异常。

大多数动态语言都可以重写“处理未知调用的方法”。这是一个很强大的技术，但在使用时需要格外小心，因为它会改变程序方法分派系统的机制。如果没有合理使用，代码会变得难以理解。

Ruby的生成器库（由Jim Weirich编写）就是一个如何正确使用method_missing的好例子。生成器库是用来生成XML标记的，它非常合理地使用了闭包和method_missing。

可以通过一个简单的例子来说明这一点。如下代码，

下载 lairs/frags

```
builder = Builder::XmlMarkup.new("", 2)
puts builder.person do |b|
  b.name("jim")
  b.phone("555-1234", "local"=>"yes")
  b.address("Cincinnati")
end
```

会生成下面这段标记。

下载 lairs/frags

```
<person>
  <name>jim</name>
  <phone local="yes">555-1234</phone>
  <address>Cincinnati</address>
</person>
```

3.8 总结

两年之前，Dave Thomas在他的博客中提及“代码拆招”（code katas）的概念：在尝试使用各种方法来解决一个简单问题的过程中，研究和比较各种解决方案的优劣。本章就是这样的一个练习。最后我也没有给出任何确定的结论，但它确实带着我们探讨和领略了用Ruby编写内部DSL的各种方法（当然，大部分方法也可以通过其他语言来实现）。

第 4 章

语言的盛景

Rebecca J. Parsons，技术总监

4.1 简介

当植物学家徜徉在草木茂盛的田野上，在惊叹于植物的多样性的同时，他很可能会暗暗鉴别所碰到的不同品种。同样地，计算机科学家们在惊异于计算机语言的多样性的同时，也会根据它们的基础特质而对不同的语言进行分类。但即便有了这些分类，深入理解这些特质也会帮助我们更好地了解那些新近出现的计算机语言。

4.2 样本

植物的基本特性包括颜色、大小、叶片形状、花朵、果实以及荆棘等物理性状，依据这些特性对植物进行分类是简单且直接的。而计算机语言的特性则涉及：可用的语句类型、如何处理类型、语言本身是如何实现以及程序基本组成原则等逻辑问题。考虑到各个特性之间的差异，一种语言可以被划分到不同的类属里，也就不足为奇了。在这篇文章里，我们将会考察一些样本，并根据这些样本为计算机语言构建一棵“生命之树”。我们既会考察新的语言也会考察旧的语言，并将格外地注意每个语言特有的受人瞩目的特性。

首先来看看古老而荣耀的Fortran。Fortran是科学计算领域的一种经典的语言，其语言特征相当简单。赋值语句通过变量名改变内存状态；其他语句访问内存状态并进行计算。根据这个特性，我们可以说，Fortran是一种典型的*命令式*语言。因为过程是组织语句的主要机制，命令式语言通常也被称作*过程式*语言。虽然我们根据这个特性对Fortran进行了分类，但是它的其他一些特性仍然值得关注，因此有必要继续考察并区分其他特性。

Fortran还是一种*静态*语言。所谓静态语言，是指程序首先需要经过编译与链接，才能被加载

到内存执行。编译器将源程序翻译为机器语言，并执行相应的优化。编译器也负责判断程序在语法上是否正确。

下来让我们看看另一个经典的程序设计语言——Lisp。虽然在相当长的一段时间内，Lisp被当作人工智能的同义词，但实际上Lisp有着更广泛的应用。虽然有人开玩笑说LISP是“许多无聊的括号”（Lots (of) insignificant silly parentheses）的缩写——类似的玩笑还有很多——但是这个语言有许多值得注意的特性。Lisp是一种*函数式*语言，而不是过程式语言。Lisp的基本编程单位是数学意义上的“函数”。“纯函数”根据传进的参数计算返回值。当给定相同的参数时，“纯函数”永远返回相同的结果。也就是说，“纯函数”没有在不同调用间可以被保存的内存数据或状态。

Lisp也是一种*动态*语言。这个语言特性主要指何时进行某些特定的决策或计算。动态语言将许多编译器执行的计算推迟到运行期。因此，静态语言中“编码、编译、测试、撞墙并重复”的开发周期，在动态语言中变成了“编码、测试、撞墙并重复”。也就是说，与静态语言不同，动态语言编制的程序直接被执行。不过，随着CLR和JVM这类虚拟机的日趋流行，这个区别也日渐模糊。但是动态语言仍然是一个相当重要的分类。

Lisp还是一种*动态类型*语言。在动态类型语言中，某个特定值的类型直到语句被执行的时候才能确定。因此，在Lisp中没有“指定某个特定变量为某个特定类型”的语法表示。Lisp中的变量与其当前所持有的值具有相同的类型。变量X在某个地方可以是整数类型，而在另外一个地方可以是一个列表或者布尔值。动态语言和动态类型语言并不是指代同一类语言。它们分别涉及了不同的语言特性，并且在实现层面上，它们也不一定有关联。

现在让我们考察一个更加现代的语言——Java。Java是一种*面向对象*语言，因此在Java程序中主要的组织单元就是对象。概念上来说，一个对象是一组状态变量和方法的集合，这些方法和状态由类来指定。相应的，对象是类的成员。同一个类的对象是相关但有所区别的实体。对于什么是面向对象语言以及其必备的特性，有许多相互冲突的定义。但是所有这些定义几乎都包含了两个特性：继承与封装。*继承*是类以及对象彼此关联的一种方式。子类从父类继承了其所有的状态和方法，同时，子类也可以通过引入新的或重写旧的状态和方法来扩展类的定义。封装是“信息隐藏”的一种实现技术。对象的实现细节应该被隐藏在接口之后，从而避免对于实现细节的依赖。*多态*是指一种根据不同的类型执行不同的函数的能力，虽然很多非面向对象语言也具有多态特性，但是它通常也被当作面向对象语言的一个必备特性。同样地，封装也不是区分面向对象语言和非面向对象语言的决定性特性，在很多非面向对象语言中也可以使用封装特性。

除了面向对象的特性，Java还具有很多与C和C++等“大括号语言”相似的构造，因此它也具有一些命令式语言的特性。因此，Java并不是一种纯粹的面向对象语言。

Ruby语言是时下IT界的宠儿。Ruby是一种面向对象的、动态以及动态类型的语言。Ruby具有一种强大的扩展机制，它对于函数式甚至过程式编程都能提供很好的支持。

下面来看看Haskell，一种并不广为人知的语言（不过它正在渐渐为人所认可）。Haskell是一种纯函数式语言。与Lisp不同，它不具有任何的命令式构造结构。它是静态类型的，使用*类型推演*来减少冗余的显式类型声明。不过Haskell与其他语言最大的不同在于它的惰性语义。*惰性语言*是一种从不对任何不必要的表达式进行求值的语言。这种惰性对于数据项同样有效，比如，如果需要某个列表的第一项，则仅仅计算第一项对应的元素。在近期提及Haskell的文章或讨论中，这种不同的执行语义的作用通常被过分地夸大了，不过它也确实使得Haskell有别于其他函数式语言，通过它编写程序也成为一种不同的锻炼。

让我们考察另外一个非常特别的例子——SQL。SQL是一种用于访问关系数据库中数据的通用查询语言。SQL是一种*声明式*语言。用声明式语言编写的程序只描述要计算什么，但不会具体指定如何去进行计算。例如在SQL中，语句描述所期望数据的特质，而不是如何找到这些数据。Prolog是另外一种为大众所熟悉的声明式语言。Prolog的程序由描述了系统状态的逻辑断言（公理与推演规则）组成。Prolog程序的执行过程主要是，根据相关状态，回答由公理和推演规则所定义的断言的逻辑可证性问题。不同于命令式语言所采用的描述实际计算的抽象计算机模型，Prolog的计算模型是一种从断言推理出结论的算法。

很多人认为声明式语言的定义并不是很有用。但是，如果这样考虑或许能够更好的利用这个定义：当你看到某个特定语句的时候，你能确定地指出它所表示计算的哪些方面？对于非声明式语言而言，程序中的语句表示了将要发生的计算；而声明式语言则更多让你无需关注计算细节的同时，了解了所希望结果的一些特性。必须承认，纵然做出这样的区分，对声明式语言的定义仍然不够清晰。另外，随着非声明式语言抽象程度的提高，以及很多编译器优化的大量使用，这种区分将变得更加模糊且无趣。

最后，让我们看看近期闪闪升起的Erlang。Erlang是一种函数式的、严格的动态类型语言。它在语言级别支持并发计算。虽然上述其他语言也可以通过线程或者添加一个消息层来支持并发执行，但是它们从语言上来讲，都是顺序语言。而Erlang程序则显式地描述程序该如何并发地执行，以及如何通过消息在各个并发部分间通信。

4.3　各种各样的分类

回顾一下我们考察过的这些语言就可以看出，对它们的分类基于几个方面：语言的组织原则、类型系统、执行行为以及具体实现。然而，还有一些相当重要的类别——至少是与通用编程语言

有关的重要类别——并不在上述列表中。任何支持条件语句及无限迭代的语言就可以被看做图灵完备语言，这类语言可以描述任意接受有限输入并在有限时间具有有限结果的程序。

“图灵完备”揭示了什么呢？它表明，无论人们如何狂热地赞美他们所喜欢的语言，所有主要的通用编程语言，包括Lisp、Ruby、Java、C#、C甚至古老而丑陋的汇编语言，都具有相同表现力。是的，我说的是相同的表现力，也就是说：没有一种语言可以写出其他语言不能表达的程序。

虽然你可以使用任意图灵完备语言编写相同含义的程序，但并不意味着你就应该这样做。不同的问题需要不同的解决方案；而不同的语言提供了不同的抽象机制和不同技术来支撑不同的解决方案。同样的算法在不同的语言中不仅看起来截然不同，实际运行效果也可能不尽相同。例如，一种语言可能使算法以更高的效率执行，而另一种语言能提供更清晰易懂的算法实现结构。编译器优化也或多或少地依赖于其所编译的语言。理解语言的不同特性及其所支持的不同编程模型，可以让我们应对特定任务时有针对性地选择更恰当的语言。下面让我们考察一下语言差异的各个方面以及可能做出的对语言的选择。

范型

一般而言，编程范型包括命令式、过程式、函数式、面向对象式、声明式以及逻辑式。有一些语言支持多种范型。比如Common Lisp，它既是一种函数式语言，也支持面向对象的某些概念。C++除了面向对象的特性之外，也支持很多过程式编程的特性。下表总结了主要编程语言的范型。

类别	定义	实例
命令式	修改内存状态的语句序列	Fortran，汇编
过程式	通过过程（即一组语句）组织程序	C, Pascal, Cobol
面向对象式	通过对象组织程序	Smalltalk，Java，Ruby
函数式	通过无状态函数组织程序	Lisp, Scheme, Haskell
逻辑式	通过公理和推演规则表述期望结果的特征	Prolog, OPS5
声明式	描述解决方法而不是如何实现解决方法	XSLT, SQL, Prolog

上述表格看起来似乎综合了一些并不太相关的东西。实际上，这个表格是综合了三种不同的特质而来的。

这三种特质是组织结构表示、状态表示和范围表示。如前所述，不同的语言范型通过不同的基本单位——对象、函数、过程甚至是一行行的语句——来组织代码。同样，不同的语言也具有

不同的状态表示。例如，命令式语言显式地修改内存状态，而函数式语言则不对内存状态进行任何修改。最后，不同的语言具有不同的范围形式——即状态可见性。比如，在面向对象语言中，状态保存在对象中，所有状态仅仅在对象内可见；命令式语言中，所有程序都可以访问全局变量所指代的状态；而在函数式语言中，所有变量仅仅在当前函数体内被绑定到具体的值，但是这些值不能被修改也不能被其他函数体所见。

类型特质

一个标识符的类型表明了这个标识符可以持有哪些值，以及可以在这个标识符上执行哪些操作。有许多不同种类的类型系统，它们分别表示了不同的类型特质。然而我们通常讨论的类型特质却主要是关于变量的类型何时被确定的。静态类型语言通常在编译期将一个单一的类型指定给变量。而动态类型语言中，变量类型则是在将要对变量执行操作的时候才会确定。不同于静态语言，在这类语言中，给定变量的类型可能会根据执行顺序的不同而有所不同。例如，Ruby中使用一类特定的动态类型被称作*duck typing*。在duck typing中，一个值的类型被弱化了，它不需要严格匹配特定的类型，当然通常我们也不需要进行这样的匹配。

另一个重要但被滥用的术语是强类型。强类型语言的一个常用定义是：若程序有可能在运行期产生类型错误的话，则它不能够通过编译。显而易见，这个定义需要另一个关于“什么是类型错误”的定义。虽然“除以0”可以当作一个类型错误，但类型错误通常是指在数组或者字符串上调用算术操作。

最后，我们看一下类型推演——这是一种有趣的类型识别方式。当使用类型推演时，编译器尝试推理出一个能够使程序正确执行的类型。虽然有那么一些毫无问题的程序，类型推演算法却无法找出合适的类型，但这种算法的存在仍然可以使得强类型语言易于使用。

下表列出了一些常见语言的类型特质：

类别	定义	实例
静态类型	变量类型在编译期确定且不可更改。	Java，C，Fortran
动态类型	变量类型在变量被访问时确定。	Scheme，Lisp，Ruby
强类型	不会发生运行时类型错误	Haskell，C++（如果忽略类型强转的话）
类型推演	使用类型推演算法而不是类型定义来确定变量的类型	Haskell，ML
Duck Typing	仅仅检查一部分需要的类型信息	Ruby

执行行为

在这里我们主要讨论两种不同的行为类别，当然其他类别也可能会有所有涉及。第一种行为区分了顺序语言和并行语言。第二种则区分了惰性语言和严格语言。

虽然客户端/服务器模型的出现引入某种形式的并发性，但大多数程序员从不关心并行计算。因此，大多数语言都是顺序语言也就不足为奇了。这些语言的执行语义总是假设在某一时刻只会有一条语句被执行。但实际的情况是，很久以来整个计算机业都在使用并行计算机，许多应用也大量依赖于并行和分布式计算。在一些情况下，并行性可由编译器进行推演；而在另外一些情况下，则需要引入消息层、任务层，以及相应的锁或者信号量机制才能提供适当的并发机制。有些语言对于并发性提供显式的支持，这些语言之前大多局限在科学研究或者复杂的金融分析领域。但随着多核处理器的普及以及应用程序用户期望的膨胀，它们开始越多越多地进入主流视野。

惰性求值在商业应用中则更加少见。但是随着越来越多的人开始关注Haskell，惰性求值的威力也越来越多被大家所了解。除惰性求值语言外，几乎所有语言都具有严格执行语义——简而言之，就是指所有语句依次执行，直至程序结束。而惰性求值在概念上则指这样一种执行语义：首先确定程序的结果是什么，以及哪些语言需要被执行才能得到这个结果，而任何与最终结果无关的语言都将被忽略。请看下面的例子（虽然有点傻）：

```
X = Y/0;
Y=6;
End(Y);
```

上例中，End语句表示最终期待的结果是Y的值。在严格语言中，这个段程序将会因为除零异常而失败（这里我们不考虑编译器对于未使用语句的优化）。而在惰性求值语言中，就算没有编译器优化，这段程序也会正常地返回结果6。因为最终结果与变量X无关，任何与X有关的语句都将被忽略。当然，惰性求值的影响远不止优化这么简单，它还有很多其他的用途。例如，它将使我们可以操作并指定可容纳无限数据的结构。不过这些已经超出了本文的讨论范围，我们就不再展开了。

在这个小节内，我们不需要一张总结语言执行行为的表格。因为除Haskell及其近亲Hope之外，大多数语言都是严格语言。虽然我们可以在严格语言内模拟惰性求值语义，但这些语言本身的语义仍然是严格的。

实现模型

最后一个类别是关于语言是如何实现的。通常的实现模型有两种：编译语言和解释语言。在很早之前，解释语言的执行速度是相当缓慢的，因此所有“在现实中被使用的系统”都需要使用

编译语言来完成——这个古老的智慧就在程序员间代代相传。解释语言被当作一种玩具语言或者是只被视为脚本语言，以至于尽管当时确有一些人使用Lisp来编写一些“在现实中被使用的系统”，但很多人并不相信这个事实。然而随着虚拟机的流行，编译和解释的界线开始变得模糊。在一些语言中（比如Java），程序被编译成某种字节码的表现形式，这些字节码随后将被虚拟机解释执行。不过纵然如此，我仍然觉得以下的区分仍是有用的：编译器根据源代码创造一些需要被执行的东西来获得源代码的结果；而解释器则根据源程序直接产生结果。

“脚本语言”这个词似乎传达了这样一种信息，这类语言不配用来编写程序，只能用来编写脚本。不过考虑到相当大量的系统都是使用脚本语言编写的，这个说法看起来相当的不明智。

还有一些语言，比如Scheme的一些版本，同时提供编译和解释的版本，进一步模糊了这个特性的分界。

同样的，这里也不需要一张表格根据这个语言特性对语言进行分类，因为很多语言同时支持编译和解释，而使用字节码的虚拟机进一步使得这个问题复杂化了。

4.4 语言的“生命之树”

下面的这张表中，总结了一些常见的语言在上述类别中的位置。虽然很多语言具有截然不同的语法，但它们却具有相似的特性。从程序设计的角度上来说，语法是无关紧要的，语言的表达能力主要来自语言的特性而不是语法。然而，对于程序员而言，他们总是挑剔语言的语法——这也是正常的，毕竟他们整天都跟语法打交道。IDE的兴起的确减轻了语法的负担，但它们不可能被IDE一劳永逸地消除。

语言	范型	类型系统	实现
汇编	命令式	动态类型	汇编，顺序的
Fortran	命令式	静态类型	编译，顺序的，有并行化编译器
C	命令式，过程式	静态类型，但带有指针	编译，顺序的
C++	命令式，过程式，面向对象	静态类型，但带有指针	编译，顺序的，有并行化编译器
Java	命令式，过程式，面向对象	静态类型	编译，使用线程进行并发设计
Lisp	函数式带一点点过程式，（CLOS引入了面向对象）	动态类型	解释，编译，顺序的

Scheme	函数式带一点点过程式	动态类型	解释，编译，顺序的
Haskell	函数式，延迟计算	静态类型，类型推演	解释，编译
Ruby	面向对象	动态类型，Duck Typing	解释，编译，顺序的
Prolog	声明式	动态类型	
Scala	函数式，面向对象	静态类型	解释，编译，有并行支持
Erlang	函数式	动态类型	解释，编译，有并行支持

这个列表是不详尽的。我们还可以讨论更多语言特性。此外，很多古老语言特有的特性开始出现在一些现代语言中（闭包，高阶函数等）。语言还会继续演化下去，不断地提供不同的抽象和方法。

4.5 这些都很有趣，但我为什么要关心

语言之争已经持续了20多年，没有理由相信它们会结束。有些人似乎认为存在有一种完美的语言，而我并不这样看。尤其是考虑到特定领域语言（DSL）的日趋流行，以及新方法（意图编程和语言工作台）的出现，对“完美语言”的追求就更显可笑。现实的情况是，不同的语言应该具有一些不同特性和语法，用以实现某种特定类型的组件或程序。显而易见，某个语言适用的场合越多，它就越有可能成为一种通用语言。当然，从需要维护很多应用程序的CIO的角度来看，使用一种语言将极大地简化用人策略——至少表面看起来是这样。我们仍会继续争论Ruby和Java的孰是孰非以及下一批竞争者是谁。不过我仍然梦想，有一天，我们谈论的是不仅仅是该使用哪种语言，而是应用哪种语言来解决哪类问题。

第5章

多语言开发

Neal Ford，意见领袖[①]

用不了十年，所有程序员都将用Smalltalk编程，不论他们把它叫什么。

——Glenn Vanderburg

时间回到1995年，当时C++程序员们还在为指针、内存管理和其他怪异的技巧而身心疲惫的时候，Java出现了。它减轻了C++程序员的痛苦，从而受到热捧。程序员可以用Java更轻松地完成工作。不过为了让Java能够更成功，Java设计者们需要吸引当时主流程序员——也就是那些C++程序员——的注意。因此，Java语言的设计者有意地让Java看起来非常像C++，这在当时看来是非常合理的。如果让开发者从最基础的东西开始学习一门新的语言，他们是很难接受的。

但是走到2008年，向后兼容性的问题已经不那么重要了。Java开发新手要学很多奇怪的内容，而且这些内容大多与需要解决的问题没有关系，而仅仅是为了满足Java中的一些规矩套路。看看下面这段很多Java开发者第一次碰到的Java代码。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

想想你要向一个开发新手解释多少事情，才能让他理解这段代码？Java自身充斥着C++的遗风（比如索引基于0的数组），以及在1995年看来合理的设计（比如区分原生数据类型和对象）——这些都成为阻碍当代程序员生产率提高的障碍了。

幸运的是，Java的设计者们在创造Java时做出了一个英明的决定：将语言和平台分离。这样，

① Meme Wrangler，ThoughtWorks公司的一种职位。——编者注

就给了开发者一个逃离Java牢笼的机会，它就是多语言开发（Polyglot Programming）。

5.1　多语言开发

单词*polyglot*是指可以讲多种语言的能力。多语言开发利用了Java中语言和平台分离的特性（C#亦如此）。开发者可以使用特定的语言解决特定的问题。现在，在Java虚拟机和.NET托管运行时上可以运行数以百计的编程语言了。然而作为开发者，我们还没有充分利用这一能力。

当然，开发者们一直都使用着多种编程语言进行开发：大多数应用程序都使用SQL访问数据库，用JavaScript为网页添加交互性，更不用说无处不在的XML配置文件。但是，多语言开发的思路与此并不相同。前面的几个例子都与JVM无关；它们运行在Java世界之外。这让人非常头疼。试想为了解决对象和基于集合论的SQL之间的阻抗不匹配，人们已经花掉了多少亿美元？这种阻抗不匹配会令开发者十分不安，因为在用到多种语言的地方，他们会感到痛苦。但是多语言开发是不同的，它利用的编程语言都会生成运行于JVM的字节码，因此不存在阻抗不匹配的问题。

多语言开发的另一个问题是你必须变换语言。在以前，变换语言通常都意味着变换平台。这对于开发者来说明显是个坏消息，因为他们不希望重写所有的库。但是对于像Java和C#这类与平台分离的语言来说，你不必再为这个问题而困惑了。多语言开发让你能够继续利用所有现有的资产，同时还能够选择更适合完成当前工作的语言。

那么，所谓“更适合完成当前工作的语言”是什么意思呢？下面几节展示了一些应用多语言开发的范例。

5.2　用 Groovy 的方式读取文件

现在有一项任务：编写程序，用来读取文本文件、打印文件的文本内容，并在每一行的前面加上行号。下面是Java代码。

下载 ./code/ford/LineNumbers.java

```
package com.nealford.polyglot.linenumbers;

import java.io.*;
import static java.lang.System.*;

public class LineNumbers {
    public LineNumbers(String path) {
        File file = new File(path);
```

```
        LineNumberReader reader = null;
        try {
            reader = new LineNumberReader(new FileReader(file));
            while (reader.ready()) {
                out.println(reader.getLineNumber() + ":"
                        + reader.readLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException ignored) {
            }
        }
    }

    public static void main(String[] args) {
        new LineNumbers(args[0]);
    }
}
```

下面是能够完成相同功能的Groovy（一种在JVM上运行的脚本语言）代码。

下载 ./code/ford/LineNumbers.groovy

```
def number=0
new File (args[0]).eachLine { line ->
    number++
    println "$number: $line"
}
```

对于简单的任务，Java显得过于臃肿和复杂了。在刚才的例子中，一板一眼的异常处理代码远多于完成功能的代码。Groovy可以为你处理大部分繁琐的工作，让你更容易获得可用的代码。这段代码会编译为Java字节码，最终结果和Java代码的效果是完全相同的。不得不承认一点，Groovy版本的字节码并不是非常高效：Groovy必须对Java的字节码进行很多处理，让它变得更加动态（比如通过代理对象调用Java类）。但是在这个简单的任务中，到底是程序员的生产率更重要，还是执行的效率更重要？如果这个完成读取文件并添加行号的任务，所用的时间是500毫秒或者100毫秒，谁在乎呢？你花在编写代码上节省下来的时间是这个时间的几百万倍。挑选一个更适合工作的工具，远比优化性能重要得多。

5.3 JRuby 和 isBlank

当然，上一节的例子中，那个简单的脚本完成的是一项Java并不擅长的任务。对于在Web应用程序中验证参数不为空这种更普遍的情况呢？

下面的Java代码来自于Apache Commons项目，它满足了很多Java基础代码的常见需求。这段代码可以判断一个字符串是否为空，几乎所有的Web应用程序都要对用户输出参数进行这项判断。于是就有了StringUtils类中的isBlank()方法。

下载 ./code/ford/StringUtils.java

```
public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
        return true;
    }
    for (int i = 0; i < strLen; i++) {
        if ((Character.isWhitespace(str.charAt(i)) == false)) {
            return false;
        }
    }
    return true;
}
```


这段代码暴露出很多Java语言固有的缺陷。首先，由于Java不允许你改变或者扩展String类，因此这个方法是在一个名为StringUtils的类中。这是一个典型的*落单方法*（poorly hung method）——它没有被放置到自己本应属于的那个类中。另一个缺陷是，你必须不断地校验传递给你的对象是否为null。null在Java中是一个特殊类型：它既不是原生数据类型，也不是对象。很多Java代码都需要校验对象是否为null。最后，你必须要遍历字符串，确保所有的字符都是空格。当然，你无法调用每个字符上的方法做去判断（因为字符是原生类型）；而是必须使用Character封装类。

下面是完成相同功能的JRuby代码。

下载 ./code/ford/blankness.rb

```
class String
  def blank?
    empty? || strip.empty?
  end
end
```

下面是证实它能正确工作的测试。

下载 ./code/ford/test_blankness.rb

```
require "test/unit"
require "blankness"

class BlankTest < Test::Unit::TestCase
  def test_blank
    assert "".blank?
    assert " ".blank?
    assert nil.to_s.blank?
    assert ! "x".blank?
  end

end
```

注意上面代码中的几件事情。第一，Ruby允许你直接为String添加方法，从而生成带有属性的方法。第二，由于你不必区分原生类型与对象，所以代码变得非常简单。第三，在测试中，我不必担心nil的情况：在Ruby中，nil也是一个对象（像这个语言中的其他东西一样），因此如果我试图传入一个nil，它的to_s()方法（Ruby版本的toString()方法）会返回一个长度为0的字符串——也就是空字符串。

你无法将这段代码在Java中重新实现，因为在Java世界中，String类是final的。但是，如果你使用基于JRuby的Ruby on Rails，就可以这样操作Java的字符串了。

5.4　Jaskell 和函数式编程

目前为止，我们看到的例子大多是关于弥补Java的语言级缺陷的。但是多语言开发还能弥补语言的基础设计决策的缺陷。比如，在Java和C#这种命令式的语言中，编写与线程相关的代码是非常困难的。你必须了解使用synchronized关键字的副作用和微妙之处，以及多线程访问共享数据时会带来哪些不同。

在多语言编程的条件下，你可以使用函数式语言，从而彻底避免这些问题。这样的语言包括Jaskell（Java版本的Haskell）以及Scala（一种为JVM编写的现代的函数式语言）。

函数式语言摆脱了很多命令式语言的限制。它们更加严格地遵循一些数学原理，例如函数式语言中的函数会像数学中的函数一样工作：输出完全依赖于输入。换句话说，函数在运行的过程中是不会改变其内部状态的。就好像调用数学函数，比如sin()，你不必担心某次调用会突然返回余弦值，因为sin()的任何内部状态都不会被改变。数学函数中的任何状态都不会在调用的过程中被修改。函数式语言中的函数（和方法）也以相同的方式工作。常见的函数式语言包括Haskell、O'Caml、SML、Scala、F#等。

尤其需要强调的是，函数式语言对于多线程的支持要比命令式语言好很多，因为它们拒绝使用状态。相对于命令式语言，其有利的一面是，使用函数式语言可以更容易地编写强壮的线程安全的代码。

下面进入Jaskell。Jaskell是Haskell语言的一个版本，运行在Java平台上。换句话说，它可以将Haskell代码转换为Java字节码。

下面是一个例子。假如你要使用Java实现一个类，它可以安全地访问数组中的一个元素。这个类的实现大致如下。

下载 ./code/ford/SafeArray.java

```
class SafeArray{
    private final Object[] _arr;
    private final int _begin;
    private final int _len;

    public SafeArray(Object[] arr, int len){
        _arr = arr;
        _begin = begin;
        _len = len;
    }

    public Object at(int i){
        if(i < 0 || i >= _len){
            throw new ArrayIndexOutOfBoundsException(i);
        }
        return _arr[_begin + i];
    }

    public int getLength(){
        return _len;
    }
}
```

你可以使用Jaskell中的*tuple*编写相同的功能。tuple本质上就是一个关联数组（associative array)。

下载 ./code/ford/safearray.jaskell

```
newSafeArray arr begin len = {
  length = len;
  at i = if i < begin || i >= len then
           throw $ ArrayIndexOutOfBoundsException.new[i]
         else
           arr[begin + i];
}
```

由于tuple本质上是关联数组，所以对`newSafeArray.at(3)`的调用就会转发到tuple的`at`部分，从而执行在这部分里定义的代码。尽管Jaskell不是面向对象的，但继承和多态等机制都可以用tuple来模拟，而且一些程序员们向往已久的功能——例如*mixin*——在Jaskell中也可以用tuple来实现，而在Java语言中就做不到。*mixin*允许你在不使用继承的情况下向一个类中注入代码（而不仅仅是增加方法签名），从而提供了一种取代接口与继承机制的可能性。

Haskell（以及Jaskell）的特性之一是函数的惰性求值。在Haskell中，不确定的计算在其真正需要前，是不会执行的。例如，下面的代码在Java中会导致问题，但是在Haskell中是完全合法的。

```
makeList = 1 : makeList
```

这段代码可以理解为“创建一个只包含单一元素‘1’的列表。如果还需要更多的元素，重复执行该操作。”这个函数最终会创建一个由‘1’组成的无限长度的列表。

假如你要实现一个复杂的调度算法，用Java可能需要1000行代码，但是Haskell只需要50行就够了。既然如此，为什么不充分利用Java平台支持多种语言的能力，用其他更适合这项任务的语言来编程呢？就像每个项目都有数据库管理员一样，我们以后还需要越来越多的具有特殊知识背景的人来编写代码，实现特定的功能。

不大可能只用Jaskell去实现一个完整的应用程序。但是，在那些大型的应用程序中，为什么不去好好地利用它的优势呢？比如说，你在开发一个Web应用程序，它需要一段高并发的调度代码。那么，可以只用Jaskell编写调度程序，用JRuby的Rails（或者Groovy on Grails）开发与Web相关的代码，最后利用已有的代码与老旧的大型机通信。由于有Java平台，你可以在字节码级别上把它们粘合到一起，从而提高你的生产率，因为你所用的工具更适合面前的问题。

5.5　Java 测试

即使你不愿意改变现在正在做的事情，也同样可以采用多语言开发。对于现有的代码，你仍然有机会采用更适合的语言。最常见的事情之一就是测试复杂的代码。Java并不具备足够的灵活性让一个对象去模仿其他对象，所以创建符合需要的模仿（mock）对象将会非常耗费大量的时间。为什么不用更有效的语言去写测试（只是测试而已）呢？

下面的实例演示了使用JMock（一个流行的Java模仿对象库）测试`Order`类和`Warehose`类（事实上是类与接口）的交互。

下载 ./code/ford/OrderInteractionTester.java

```
package com.nealford.conf.jmock.warehouse;

import org.jmock.Mock;
import org.jmock.MockObjectTestCase;

public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new OrderImpl(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");
```

```
    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

}
```

这段代码测试了Order类与Warehouse类（通过其接口）之间的交互，并且验证了相关的方法是否被调用，以及结果是否正确。

下面是利用JRuby（以及Ruby世界里强大的模仿对象库——Mocha）完成的相同的测试。

下载 ./code/ford/order_interaction_test.rb

```ruby
require 'test/unit'
require 'rubygems'
require 'mocha'

require "java"
require "Warehouse.jar"
%w(OrderImpl Order Warehouse WarehouseImpl).each { |f|
   include_class "com.nealford.conf.jmock.warehouse.#{f}"
 }

class OrderInteractionTest < Test::Unit::TestCase
  TALISKER = "Talisker"

  def test_filling_removes_inventory_if_in_stock
    order = OrderImpl.new(TALISKER, 50)
    warehouse = Warehouse.new
    warehouse.stubs(:hasInventory).with(TALISKER, 50).returns(true)
    warehouse.stubs(:remove).with(TALISKER, 50)

    order.fill(warehouse)
    assert order.is_filled
  end

end
```

由于Ruby是动态语言，因此这段代码看上去会更加简单。JRuby会将Java对象封装在代理类中，你能够直接实例化一个接口（本例中为Warehouse）。另外，你只要在测试的开始加上require 'warehouse.jar'，就可以将所有相关的Java类导入在测试路径上了。在Java里，你不希望能这么做么？

多语言开发不一定会给你当前的工作带来天翻地覆的冲击。在大多数公司里，测试代码并不是“官方的”代码，所以甚至有可能不用获取许可就能使用JRuby开始编写测试。

5.6　多语言开发与未来之路

在2007早些时候，ThoughtWorks发布了它的商业产品处女作——Mingle，一个敏捷项目管理工具。当时，对于Mingle来说，能否尽快上市是至关重要的，所以我们决定使用Ruby on Rails进行开发。但是，我们也希望能够重用一些已经存在的库，包括Subversion支持库和一个基于Java的图表库。因此，我们最终选择用Rails on JRuby来实现：它既允许我们使用已有的Java库，又能发挥Ruby快速开发的优势。Mingle充分体现了多语言开发的精神：一方面选择使用针对当前工作的最佳工具，另一方面发挥底层平台的健壮与资源丰富的优势。

将各种解决方案强行塞入单一编程语言的日子即将一去不复返。既然有出色的托管运行时（Java和CLR），我们就应该充分利用这些平台，同时选择更好的工具。利用多语言开发，你既可以混合多种适合不同情形的语言，又不必丢弃已经存在的代码——毕竟它们还是非常重要的。在Java和.NET这两个已获得充分验证的平台上，新语言的开发正在迅猛发展。作为一名开发者，你需要学习如何利用这一发展的优势，这样才能使用更适合工作的工具编写更棒的代码。

第 6 章

对象健身操

6

Jeff Bay，技术负责人

6.1 九步迈向优秀软件设计

我们大都读过糟糕的代码。这些代码通常都难以理解、测试和维护。面向对象编程一直在鼓励我们摆脱过程式代码的桎梏，帮助我们以累加的方式编写代码，并重用已有的组件。但有些时候，我们似乎只是把原来陈旧的、充满耦合的设计从C程序搬到了Java中而已。这篇文章会向编程新手讲述一些非常实用的最佳实践。对于资深的程序员来说，这篇文章可以让你再回顾一下以往的最佳实践，也可以在培训新员工时，作为抽象原则的具体实现示例。

要想掌握优秀的面向对象设计并非易事；但一旦掌握，优秀的设计会在简单性上带来巨大的回报。从过程化的开发到面向对象设计之间的思维转换，要远比看上去的复杂得多。很多开发者都自以为擅长OO设计，但在实际应用时，还是不自觉地回到过程化的编程习惯上了，这已经是根深蒂固的观念。更糟糕的是，很多实例和最佳实践（甚至Sun在JDK中的代码）甚至鼓励人们采用糟糕的OO设计——其中一些尚有性能考量作为托辞，而另一些则纯粹是历史包袱。

优秀设计背后的核心概念其实并不高深。比如内聚性、松耦合、零重复、封装、可测试性、可读性以及单一职责。这七条评判代码质量的原则就已经被广泛接受了。然而真正困难的是如何把这些概念付诸实践。理解了封装就是隐藏“数据、实现细节、类型、设计或者构造”，这只是设计出良好封装的代码的第一步而已。因此，本文接下来是一系列实践规则和练习，它可以帮助你将良好的面向对象设计原则变得更加具体，从而在现实世界中应用那些原则。

6.2 练习

在一个简单的项目里尝试一些比以前严格得多的编码标准。在这篇文章中，你会看到我给出

的“九诫”，它们会迫使你更为严格地以面向对象的风格编写代码。在日常工作中遇到问题时，这些法则可以帮你做出更正确的决定，或者给你更多更好的选择。请你暂时放弃怀疑，在一个代码量为1000行左右的小项目里严格地遵守这些法则，到时你会体验到一种完全不同的软件设计的方法。在写完这1000行左右的代码后，练习就结束了，你就能放心地将这些规则当作指导原则了。

下面的练习有一定困难，尤其因为所有规则并不是放之四海皆准的法则。比如，很难保证所有的类的长度都不超过50行，例外总会出现。但是这种做法背后的价值在于，你应该把与某一职责相关的代码移到真正的、一流的对象中。这个练习的真正价值也在于此，即引发你在开发过程中进行这种思考。所以，现在开始扩展你想象力的极限吧，然后看一看你是不是已经以一种全新的方式思考你的代码了？

规则

下面是练习中要遵守的规则：

1. 方法只使用一级缩进。
2. 拒绝使用`else`关键字。
3. 封装所有的原生类型和字符串。
4. 一行代码只有一个“.”运算符。
5. 不要使用缩写。
6. 保持实体对象简单清晰。
7. 任何类中的实例变量都不要超过两个。
8. 使用一流的集合。
9. 不使用任何Getter/Setter/Property。

规则1：方法只使用一级缩进

你是否曾经盯着一个体型巨大的老方法而感到无从下手过。庞大的方法通常缺少内聚性。一个常见的原则是将方法的行数控制在5行之内，但是如果你的方法已经是一个500行的大怪兽了，想要达到这一原则的要求是非常痛苦的。其实，你不妨尝试让每个方法只做一件事——每个方法只包含一个控制结构或者一个代码块。如果你在一个方法中嵌套了多层控制结构，那么你就要处理多个层次上的抽象，这意味着同时做多件事。

如果每个方法都只关注一件事，而它们所在的类也只做一件事，那么你的代码就开始变化了。由于应用程序中的每个单元都变得更小了，代码的可重用性开始指数增长。一个100行的，肩负

五种不同职责的方法很难被重用。如果一个很短的方法在设置了上下文后，能够管理一个对象的状态，那么它可以应用在很多不同的上下文中。

利用IDE提供的“抽取方法”功能，不断地抽取方法中的行为，直到它只有一级缩进为止。请看下面的实例。

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                for(int i = 0; i < 10; i++) {
                        for(int j = 0; j < 10; j++)
                                buf.append(data[i][j]);
                        buf.append("\n");
                }
                return buf.toString();
        }
}


Class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                collectRows(buf);
                Return buf.toString();
        }

        Void collectRows(StringBuffer buf) {
                For(int I = 0; I < 10; i++)
                        collectRow(buf, i);
        }

        Void collectRow(StringBuffer buf, int row) {
                For(int I = 0; I < 10; i++)
                        Buf.append(data[row][i]);
                buf.append("\n");
        }
}
```

注意这项重构还能带来另一种效果：每个单独的方法都变得更简单了，同时其实现也与其名称更加匹配。在这样短小的代码段中查找bug通常会更加容易。

第一条规则在此接近尾声了。我还要强调，你越多实践这条规则，就会越多地尝到它带来的甜头。当你第一次尝试解决前面展示的那一类问题时，可能不是非常熟练，也未必能获得很多收获。但是，应用这些规则的技能是一种艺术，它能将程序员提升到一个新的高度。

规则2：拒绝else关键字

每个程序员都熟知`if/else`结构。几乎每种语言都支持`if/esle`。简单的条件判断对任何人来

说都不难理解。不过大多数程序员也见识过令人眩晕的层层嵌套的条件判断，或连绵数页的case语句。更糟糕的是，在现有的判断条件上加一个新的分支通常是非常容易的，而将它重构为一个更好的方式的想法却罕有人去提及。条件判断结构通常还是重复代码的来源。例如，状态标识经常会带来这样的问题。

```
public static void endMe() {
if (status == DONE) {
        doSomething();

} else {
        <other code>
        }
}
```

你有很多种方式重写这段代码，去掉else关键字。例如下面的代码。

```
public static void endMe() {
        if (status == DONE) {
                doSomething();
                return;
        }
        <other code>
}

public static Node head() {
        if (isAdvancing()) { return first; }
        else { return last; }
}

public static Node head() {
        return isAdvancing() ? first : last;
}
```

在上面的例子中，第二段代码由于使用了三元运算符，所以代码长度从四行压缩到了一行。需要小心的是，如果过度使用“提前返回”，代码的清晰度很快会降低。《设计模式》[GHJV95]一书中关于策略模式的部分里有一个实例，演示了如何使用多态避免根据状态进行分支选择的代码。如果这种根据状态进行分支选择的代码大量地重复，就应该考虑使用策略模式了。

面向对象编程语言给我们提供了一种更为强大的工具——多态。它能够处理更为复杂的条件判断。对于简单的条件判断，我们可以使用“卫语句”和“提前返回”替换它。而基于多态的设计则更容易阅读与维护，从而可以更清晰地表达代码的内在意图。但是，程序员要做出这样的转变并不是一帆风顺的。尤其是你的代码中可能早已充斥了else。所以，作为这个练习的一部分，你是不可以使用else的。在某些场景下可以使用Null Object模式，它会对你有所帮助。另外还有很多工具和技术都可以帮助你甩掉else。试一试，看你能提出多少种方案来？

规则3：封装所有的原生类型和字符串

整数自身只代表一个数量，没有任何含义。当方法的参数是整数时，我们就必须在方法名中描述清楚参数的意思。如果此方法使用“Hour”作为参数，就能够让程序员更容易地理解它的含义了。像这样的小对象可以提高程序的可维护性，因为你不可能给一个参数为“Hour”的方法传一个“Year”。如果使用原生变量，编译器不能帮助你编写语义正确的程序。如果使用对象，哪怕是很小的对象，它都能够给编译器和其他程序员提供更多的信息——这个值是什么，为什么使用它。

像Hour或Money这样的小对象还提供了放置一类行为的场所，这些行为放在其他的类中都不合适。在你了解了关于getter和setter的规则时，这一点会非常明显，有些值只能被这些小对象来访问。

规则4：一行代码只有一个“.”运算符

有时候我们很难判断出一个行为的职责应该由哪个对象来承担。如果你看一看那些包含了多个“.”的代码，就会从中发现很多没有被正确放置的职责。如果代码中每一行都有多个“.”，那么这个行为就发生在错误的位置了。也许你的对象需要同时与另外两个对象打交道。在这种情况下，你的对象只是一个中间人；它知道太多关于其他对象的事情了。这时可以考虑把该行为移到其他对象之中。

如果这些“.”都是彼此联系的，你的对象就已经深深地陷入到另一个对象之中了。这些过量的“.”说明你破坏了封装性。尝试着让对象为你做一些事情，而不要窥视对象内部的细节。封装的主要含义就是，不要让类的边界跨入到它不应该知道的类型中。

迪米特法则（The Law of Demeter，“只和身边的朋友交流”）是一个很好的起点。还可以这样思考它：你可以玩自己的玩具，可以玩你制造的玩具，还有别人送给你的玩具。但是永远不要碰你的玩具。

```
class Board {
    ...

    class Piece {
        ...
        String representation;
    }
    class Location {
        ...
        Piece current;
    }
```

```
    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            buf.append(l.current.representation.substring(0, 1));
        return buf.toString();
    }
}

class Board {
    ...

    class Piece {
        ...
        private String representation;

        String character() {
            return representation.substring(0, 1);
        }

        void addTo(StringBuffer buf) {
            buf.append(character());
        }
    }
    class Location {
        ...
        private Piece current;

        void addTo(StringBuffer buf) {
            current.addTo(buf);
        }
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            l.addTo(buf);
        return buf.toString();
    }
}
```

注意在这个例子中，算法的实现细节被过度地扩散开了。程序员很难看一眼就理解它。但是，在为Piece转化成Character的行为创建一个具有名称的方法后，这个方法的名称和作用就相当一致了，而且被重用的机会也非常高——令人费解的`representation.substring(0, 1)`调用可以全部被这个具有名称的方法所代替，程序的可读性又迈进了一大步。在这片新天地里，方法名取代了注释，所以，值得花些时间为方法取一个有意义的名字。理解并写出这种结构的程序并不困难，你只需要使用一些稍微不同的手段而已。

规则5：不要使用缩写

我们总会不自觉地在类名、方法名或者变量名中使用缩写。请抵制住这个诱惑。缩写会令人

迷惑，也容易隐藏一些更严重的问题。

想一想你为什么要使用缩写。因为你厌倦了一遍又一遍地敲打相同的单词？如果是这种情况，也许你的方法调用得过于频繁，你是不是应该停下来消除一些重复了？因为方法的名字太长？这可能意味着有些职责没有放在正确的位置或者是有缺失的类。

尽量保持类名和方法名中只包含一到两个单词，避免在名字中重复上下文的信息。比如某个类是`Order`，那么方法名就不必叫做`shipOrder()`了，把它简化为`ship()`，客户端就会调回`order.ship()`——这能够简单明了地说明代码的意图。

在这个练习中，所有实体对象的名称都只能包含一到两个单词，不能使用缩写。

规则6：保持实体对象简单清晰

这意味着每个类的长度都不能超过50行，每个包所包含的文件不超过10个。

代码超过50行的类所做的事情通常都不止一件，这会导致它们难以被理解和重用。小于50行代码的类还有一个妙处：它可以在一屏幕内显示，不需要滚屏，这样程序员可以很容易、很快速地熟悉这个类。

创建这样小的类会遇到哪些挑战呢？通常会有很多成组的行为，它们逻辑上是应该在一起的。这时就需要使用包机制来平衡。随着类变得越来越小，职责越来越少，加之包的大小也受到限制，你会逐渐注意到，包中的类越来越集中，它们能够协作完成一个相同的目标。包和类一样，也应该是内聚的，有一个明确的意图。保证这些包足够小，就能让它们有一个真正的标识。

规则7：任何类中的实例变量都不要超过两个

大多数的类应该只负责处理单一的状态变量，有些时候也可以拥有两个状态变量。每当为类添加一个实例变量，就会立即降低类的内聚性。一般而言，编程时如果遵守这些规则，你会发现只有两种类，一种类只负责维护一个实例变量的状态；另一种类只负责协调两个独立的变量。不要让这两种职责同时出现在一个类中。

敏锐的读者可能已经注意到了，规则3和规则7其实是相同问题的不同表述而已。在通常的情况下，对于一个包含很多实例变量的类来说，很难拥有一个内聚的、单一的职责描述。

我们来仔细分析下面的示例。

```
    String first;
    String middle;
    String last;
}
```

这个类可以被拆分为两个类。

```
class Name {
    Surname family;
    GivenNames given;
}

class Surname {
    String family;
}

class GivenNames {
    List<String> names;
}
```

注意思考这里是如何分离概念的，其中姓氏（family name）是一个关注点（很多法律实体约束中需要用到），它可以和其他与其有本质区别的名字分开。GivenName对象包含了一个名字的列表。在新的模型中，名称允许包含first，middle和其他名字。通常，对实例变量解耦以后，会加深理解各个相关的实例变量之间的共性。有时，几个相关的实例变量在一流的集合中会相互关联。

将一个对象从拥有大量属性的状态，解构成为分层次的、相互关联的多个对象，会直接产生一个更实用的对象模型。在想到这条规则之前，我曾经浪费过很多时间去追踪那些大型对象的数据流。虽然我们可以理清一个复杂的对象模型，但是理解各组相关的行为并看到结果是一个非常痛苦的过程。相比而言，不断应用这条规则，可以快速将一个复杂的大对象分解成为大量简单的小对象。行为也自然而然地随着各个实例变量流入到了适当的地方——否则编译器和封装法则都不会高兴的。当你真正开始做的时候，可以沿着两个方向进行：其一，可以将对象的实例变量按照相关性分离在两个部分中；另外，也可以创建一个新的对象来封装两个已有的实例变量。

规则8：使用一流的集合

应用这条规则的方法非常简单：任何包含集合的类都不能再包含其他的成员变量。每个集合都被封装在自己的类中，这样，与集合相关的行为就有了自己的家。你可能会发现作用于这些集合的过滤器将成为这些新类型中的一部分，或是根据它们自身的情况包装为函数对象。另外，这些新的类型还可以处理其他任务，比如将两个集合中的元素拼装到一起，或者对集合中的元素逐一施加某种规则等。很明显，这条规则是对前面关于实例变量规则的扩展，不过它自身也有非常重要的含义。集合其实是一种应用广泛的原生类型。它具有很多行为，但是对于代码的读者和维护者来说，与集合相关的代码通常都缺少对语义意图的解释。

规则9：不使用任何Getter/Setter/Property

上一条规则的最后一句话几乎可以直接通向这条规则。如果你的对象已经封装了相应的实例

变量，但是设计仍然很糟糕的话，那就应该仔细地考察一下其他对封装更直接的破坏了。如果可以从对象之外随便询问实例变量的值，那么行为与数据就不可能被封装到一处。在严格的封装边界背后，真正的动机是迫使程序员在完成编码之后，一定要为这段代码的行为找到一个适合的位置，确保它在对象模型中的唯一性。这样做会有很多好处，比如可以很大程度地减少重复性的错误；另外，在实现新特性的时候，也有一个更合适的位置去引入变化。

这条规则通常被描述为“讲述而不要询问”（“Tell，don’t ask”）。

6.3　总结

前面九条规则中，有八条都是非常简单易行的，能够帮助程序员获得面向对象程序设计的圣杯——数据封装。另外，有一条规则（不要使用`else`，尽量简化条件判断逻辑）鼓励程序员适当地使用多态；还有一条规则是命名策略，鼓励程序员使用一致的、直接的命名标准，不要使用难以理解的缩写。

一言以蔽之，就是想尽办法消除代码中的重复。我们每天都要面对复杂的问题，而我们的目标是，用精炼的代码表达出简单而优美的抽象。

时间长了以后，你会发现在某些情境下，这些规则其实会彼此抵触，如果强加使用可能会产生糟糕的结果。然而，出于练习的目的，你不妨花上20个小时，在100%地遵守这些规则的前提下，编写1000行代码。这样你会发现，自己已经开始打破旧有的习惯了，并且改变了一些曾经墨守的游戏规则。如果遵循本文介绍的这些规则，你必然会遇到这种情形：编程时似乎看到了一个很明显的做法，但是并不应该采纳它，因为这种顺理成章的做法很可能是错误的。

将这些规则当作纪律来遵守，它们会帮助你解答更复杂的问题，同时加深你对面向对象编程的理解程度。如果按照这些规则的要求写上1000行代码的话，你会发现最终的结果和编写代码以前所期望的完全不同。尝试一下这些规则，看看最终的结果如何。如果还能够继续遵守它们，那么以后你就能非常自然地写出符合上面规则的代码了，而不必刻意地记住这些规则。

最后要说的是，有些人可能认为这些规则过于刻板，不可能在真实的系统中实施。他们都错了——在本书即将出版的时候，我们刚刚完成了一个超过100 000行代码的系统，它严格遵守本文中提到的所有规则。参与这个项目的程序员全部都认真地遵守了上述规则。最终每个人都非常高兴地看到：如果努力地拥抱简单性，那么开发过程会快乐得多。

第 7 章

迭代经理是什么角色

Tiffany Lentz，项目经理

行业日新月异，敏捷、迭代式和迭代这些热门词已是“飞入寻常百姓家”，一个定义模糊的新角色——迭代经理，也浮出水面。这是新一代的项目经理么？抑或是美其名的团队带头人？又或者是管理上的一个新阶层？谁会被冠以这个“经理”头衔？

本文将着重阐述迭代经理作为软件团队成员的工作内容和价值。我们将分析迭代经理的职责范围，同时讨论作为一个不可或缺的角色，迭代经理在面对组织和文化挑战的情况下，如何维持一个健康的工作环境。

7.1 什么是迭代经理

通常在大型的敏捷团队里面，项目经理不可能既关注项目团队每次迭代的成功，又关注整个项目最终的成功。2000年，有一个项目团队困扰于如何甄选高优先级的工作，最终的解决办法是选出一个人以稳定的节奏给交付团队提供持续的高优先级的功能“流”：这就是迭代经理（IM）的雏形。

在迭代式开发的世界里，总是需要有人对项目团队提供支持，促进与业务客户的日常交流，使整个团队保持关注于高优先级的工作。ThoughtWorks的高级架构师Fred George把迭代经理描述成“面向内部的管理角色。迭代经理负责保证故事在团队中流动的顺畅，这牵涉到合理分配任务、在技能需要改变的时候建议更换团队成员。”

7.2 怎样成为好的迭代经理

迭代经理可以来自不同的能力背景——可以是技术能力（加上很强的人际能力！），也可以是

分析能力（加上很强的人际能力！），他们也可以从业务专家或者行政专家（加上很强的人际能力！）里面产生。他们必须拥有前瞻思考、乐观进取的态度和应对变化的才能。每天，这些面向内部的推动者使用他们的才能协助交付团队使之逐步完善。

比如，一旦明确了迭代的工作量，迭代经理就需要在迭代的整个过程中跟踪团队的进展，从实际出发，积极地在团队内部贯彻流程的改进。想象一下在站立会议上面，迭代经理听到开发人员说他已经在一个故事上面工作了三天，而这个故事原本估计是一天的工作量。因为要对每个团队成员的日常活动和迭代的进度负责，迭代经理需要去挖掘这个被低估故事的细节。如果迭代经理不能很快判断故事的真实状态，并立即与客户就迭代计划的潜在变更进行沟通，团队就有可能面临承诺失信的风险。迭代经理可以从询问如下问题开始。

- 开发人员弄清楚了故事的范围吗？
- 故事任务在最初的评估之后是否发生了变化？如果发生了，是如何变化的？
- 开发人员是否需要业务分析人员或者客户的帮助，以更好地理解故事所要求的完成结果？
- 开发人员是否需要向技术负责人寻求帮助？
- 是否有什么事情阻碍了开发人员完成故事（换句话说，是否存在硬件、软件，甚至基础设施的问题）？
- 开发人员是否被分派了另外一个项目，或者参加了太多琐碎的会议导致无法完成故事？

上述问题只是一个例子，说明为了保持团队按照预定的进度前进，以及为了向客户汇报项目的每日状态，迭代经理可能要承担的工作。每天，迭代经理都必须倾听团队的需要并且作出回应。其主要职责就是培养一台润滑良好的“机器”，能依据要求的质量在项目范围内交付功能。

迭代经理应该在技术熟练度和业务知识之间达到一个平衡。Poppendieck夫妇（Mary和Tom）写道，敏捷领袖应该“同时对客户和技术都具有深刻的理解，这样才能赢得开发团队的尊重。”良好的沟通技巧是必须的。迭代经理的职责之一就是维护开发团队与客户，以及与管理阶层之间的关系。

同时，迭代经理必须推动、主张和保障团队成员的权利。对于很多敏捷团队，这些权利来自于开发人员的“权利法案”[1]，经过了整个团队的同意。通常迭代经理都需要协助团队以确保这些权利得到了主张。

[1] 关于敏捷项目中客户、程序员和管理者三方的权利，可以参考“Extreme Programming‘Bill of Rights’”一文：http://articles.techrepublic.com.com/5100-10878_11-5121760.html。——译者注

通常这都是以在团队内部以及团队之间加强交流的方式出现。大部分的开发人员都不习惯于直接与客户交流，或者对故事的完成程度给出直接的判断。迭代经理通常需要提供数据、图表和图形的示例来推动开放的交流。

迭代经理也必须维护客户的权利。每次这样的诱惑——团队成员不按照优先级高低的顺序进行开发——初露端倪，迭代经理就要作为客户代理人出现。还记得么，客户有权利只为那些符合他们所期望的优先级顺序的工作买单？整个过程自始至终，迭代经理都必须保持一个中立的态度。

7.3　迭代经理不做什么

迭代经理（IM）并不是项目经理（PM）。与项目经理不同，迭代经理需要在工作第一线，与团队成员一起面对每日的工作活动。如果你是迭代经理，就把项目预算、资源管理、承诺以及大层面上的问题留给项目经理，而自己只关注团队！

此外，迭代经理只是一名项目成员，而不是人力经理或者资源经理。迭代经理不负责给团队成员写年度总结。这可能会影响他们的中心任务，即保持一个中立的态度——既维护团队的权利，又保持团队关注于对客户优先级最高的功能。团队成员不应该绞尽脑汁以求给迭代经理留下好印象，而应该是在需要帮助的时候要求迭代经理给予帮助。

迭代经理也不是客户。通常由团队中的商务分析师或者架构师扮演客户，这取决于故事的性质，或者真实客户的可获性。但是，迭代经理不应该扮演客户，如果他们作为客户来下决定，就无法帮助团队合理地解决问题。

最后，迭代经理不保证技术的完整性，也不保证对标准的遵守，抑或提供技术基础支持（例如，对构建、部署或者数据库的支持）。项目的实现活动，比如协调多个项目、协调部署或者演示，通常是技术负责人或者首席商务分析师来处理。

7.4　迭代经理与团队

虽然没有明确规定，但是迭代经理要承担一些日常职责。下面列举了其中一些：

- 收集花在故事开发上的时间；
- 使交付过程中的瓶颈显现出来；
- 向客户汇报团队状态；

- 解决在每日站立会议上提出的问题、阻塞和障碍；
- 控制所有流向团队的工作，管理工作的分派以保持一个可持续的速度。

收集单个故事的实际所花时间可以得到很多测量指标。收集这些时间，和其他不同的数据点比较，有助于迭代经理提高团队的产出。首先，比较迭代内已完成故事的实际所花时间和故事的点数，迭代经理就可以知道团队有多少时间花在真正的交付故事上面，又有多少花在团队会议和其他活动上面。其次，比较项目已完成故事的实际所花时间和团队计划的项目时间，迭代经理就可以明白团队的产能，以及团队对于项目的可用度。最后，比较已完成故事的实际所花时间和故事的预估时间，可以得到故事评估的准确度。上述所有的测量指标在不同的环境下都很有用，迭代经理用它们来帮助团队形成一个稳定的交付速度。

稳定的交付速度是计算未来迭代团队产能的基础。有了团队在每次迭代里面经过充分测试的产出物和每位团队成员的计划可用率，迭代经理可以根据实际的已验证的数据来规划团队的产能。产能是不受团队支配的，也不会因为明确的交付时间就自动达到。产能是预先计划的，这样团队成员才能进行自我管理。如果交付速度与业务需求不同步，可以调整其他的项目杠杆，但仍然需要使用实际的产出物来预测将来的产能。

可以借助很多测量指标和精心排列的故事卡片墙识别出项目的瓶颈。假设一个故事预估的工作量是一天，但在经过三天开发以后，却还摆放在故事卡片墙的开发栏里面：这说明遇到了瓶颈，需要团队一起讨论。Fred George创造了一种成功的测量指标，就是手指图（finger chart）。这种图表使用了堆积面积图，每块区域分别代表了迭代周期中的一个阶段。每天更新故事的状态，团队可以观察到图上每个区域的增长，以及交付周期中故事的流转。当图上的所有区域成比例地增长时，这些区域就能形成手指的形状。当图上某块区域与其他相比不成比例时（比如说“等待开发”区域比“开发”区域宽），团队能发现瓶颈的存在。此时，团队可以讨论如何消除瓶颈以使团队的交付速度回复稳定。

在每日站立会议上面，迭代经理排除掉不必要的干扰，保持团队成员使用正确的方式：过去24小时做了什么，接下来的24个小时准备做什么，遇到了什么障碍。迭代经理留心倾听每人的任务项和当天需要排除的障碍，这样团队成员可以完成故事。如果有人在每日站立会议上霸占时间，谈论标准汇报内容之外的东西，迭代经理需要带领团队重新回到关注点上面。通常的做法是建议那人在站立会议之后做一个较大的问题解答。

7.5 迭代经理与客户

正如前面讨论的，测量指标可以帮助迭代经理判断团队可持续的速度。这允许团队定期做出

承诺，并最终兑现。但是，为了团队成员能维持承诺，迭代经理必须保持客户在迭代阶段不去更改故事。迭代经理要像守门人，帮助客户对即将来临的工作排列优先级，而不是经常更改优先级使得团队无所适从。

作为守门人，迭代经理保护团队不受分心之扰，也防止客户不经意间影响团队的生产率。在迭代之外，客户可以而且应该不断地更改优先级。直到迭代开始之前，所有影响决定的因素都是可以改动的，而且要经常介绍新的信息。

项目中的“即时决定”并不是一个全新的概念，由丰田知识工程发展而成的精益开发系统在多方案同步进行的开发工程方面已经有多年的成功经验。多方案同步进行的开发工程被描述成“很谨慎地延迟决定，直到必须做出决定；努力维持各种可能的选项，尽可能延迟决定至开发团队收集到足够的决策支持信息（而不是为了所谓的果断而快速排除其他选项），从而更快地达到最优方案。”

7.6　迭代经理与迭代

迭代经理也有与迭代相关的职责。迭代经理与客户、团队一起工作，对每次迭代做出计划，包括：

- 帮助客户排列优先级；
- 整理协调团队成员的建议；
- 计划团队的交付能力。

迭代经理指导、鼓励团队，并会激发团队的士气。迭代经理通过健康检查来保持团队的坦诚。这些检查不是用来确保团队符合敏捷的所有方面，而是看团队能从敏捷提供的哪些技术中受益。

迭代经理肩上最后一个与迭代相关的职责是主持会议。迭代经理负责并引导计划会议，包括迭代计划会议和发布计划会议。适当的促进迭代计划会议和发布计划会议可以指引团队走向成功。测量指标、进行中的各项事宜，以及其他不像计划那么顺利的事情，都必须开诚布公地进行讨论。

在发布计划会议上，迭代经理展现出他们的洞察力，与客户对下一个发布需要交付的高层次功能特性做出计划。一旦计划得到认可，而且客户认同计划可能改变，迭代经理会帮助团队进行高层次的工作量评估（例如，故事的点数），让客户知道下一个发布里面会交付哪些东西。

在迭代计划会议上面，迭代经理常常要防止团队成员接受超出他们交付能力范围的工作。同

时，通过复审测量指标，迭代经理帮助团队成员学会使用相关工具，从而改进最后的会议结果。

最后，迭代经理负责推动回顾会议，这样团队能“快速发现做得不对的地方”，找出需要在下次迭代里面进行改进的地方。迭代经理需要引导团队的讨论，使其关注于本次迭代里做得好以及做得不好的地方，不时提醒成员重点是如何改进做得不好的事情。这能营造出一个负责任的氛围，也能让团队成员提升自己。

7.7 迭代经理与项目

如前所述，迭代经理承担了一系列项目相关的职责，但有时他们也会被要求负责团队文化建设。在让商业客户满意的同时，迭代经理促进形成培养满足需求、身心愉悦、工作高效和受人尊重的团队成员的环境。Fred George说，“作为次要目标，我期望看到因为迭代经理的工作，团队成员在项目结束的时候变得更为优秀。团队内充满信任，持续提高技能——这是迭代经理的工作。”

迭代经理致力于创造一个专业和敢于承担责任的氛围。这样的氛围体现在恰当的行为和习惯上，如下所示：

- 对自己、他人和客户互相尊重；
- 庆祝成功；
- 失败乃成功之母。

迭代经理要争取把团队拧成一根绳，让所有的成员能同舟共济。

7.8 总结

构建一台“润滑良好”的交付“机器”，持续地补充新的故事，调整机器——这构成了一份全职的工作。沟通清晰、实用主义至上以及处理变更的能力属于不同的技能，需要挖掘和培养。2000年之后，ThoughtWorks培养了很多迭代经理，他们在客户现场有效提高和巩固了敏捷项目的成功。迭代经理留下了可复用的流程，可以用以改进敏捷团队，完成设定故事范围的项目和改进团队文化。其最大成就是让团队成员更愉快、更高效。

每日沟通、消除不和谐、保持客户对项目最新进展的了解，可以花上开发人员整天的时间，导致几乎没有时间编写代码。如果敏捷团队里面不存在迭代经理，团队很可能会失败。团队需要一直关注手头的任务（或者故事），而把这些杂事留给迭代经理。

第 8 章

项目生命体征

Stelios Pantazopoulos，迭代经理

在医疗领域，医生或者护士可以走到病人床前，观察病人的健康状况图表，得到病人的实时生命体征数据。通过这些数据，可以快速地判断病人的健康状况，并决定是否需要进一步的治疗。如果定制软件开发项目能像诊疗者那样拿到一个实时生命体征图表，岂不是非常棒？本文主要讨论如何获得这样一个接近于实时的项目生命体征数据，并及时通知项目团队成员和利益关系人。这样，团队就可以来初步判断项目的健康状况，并采取改正措施来处理导致项目健康状况下降的根源。

8.1 项目生命体征

项目生命体征是一些可收集到的定量度量数据，用于及时反映项目的整体运行状况。这些体征包括。

- 项目范围增量（Scope burn-up）：对于某期限时所需要交付的项目范围情况。
- 交付质量（Delivery quality）：最终交付的项目状况。
- 预算燃尽（Budget burn-down）：根据项目范围交付状况统计的预算使用情况。
- 实际开发状态（Current state of implementation）：已交付的系统功能情况。
- 团队的感觉（Team perceptions）：团队对项目状态的看法。

8.2 项目生命体征与健康状况

项目生命体征是一组独立的度量数据，与项目健康不同。项目健康是根据项目生命体征的分析，对项目状态的总体判断。因此，“项目是否健康”是主观的，无法度量的。对于同一个项目生命体征，团队的两个成员对项目健康状况有不同的理解和结论。比如，项目经理可能更注重预

算的使用情况，QA（质量保证）人员可能更强调于软件的交付质量，而开发人员可能更关注范围的显著增加。同时，“项目是否健康”与每个团队成员的看法有直接关系，所有人的看法都是相互关联且同样重要的，也是独立的。

对于团队来说，得到整体项目健康状态的最好办法就是收集并展示项目生命体征。如果没有这些体征数据做为参考的话，对于项目健康状况的判断只能说是“猜测”而已。

每个团队都要定义自身项目的项目健康状况。出于一致性考虑，团队成员要列出那些他们认为应该作为项目生命体征的信息。一旦识别出这些项目生命体征后，就要定义信息指示器来显示它们。

8.3　项目生命体征与信息指示器

信息指示器（Alistair Cockburn提出的一个术语）是一个用于公开信息的显示板，向大家表明项目的当前状况。它是显示项目生命体征的一个有效方法。对于项目生命体征来说，没有哪种信息指示器是“必须品”。本文建议，对每一被实践证明的确有效的体征都要有一个信息指示器。（当然，这并不代表该方法是显示项目生命体征的唯一方法。）

8.4　项目生命体征：项目范围增量图

项目范围增量图代表项目到最后期限为止所需要交付的范围状态。该度量应该表明项目范围的数字表示、范围完成的比率、以及交付的最后期限。

范围增量的信息指示器示例

图8-1中的范围增量图用于度量并显示系统已经完成多少，还有多少没完成。图中的信息包括：

- 范围的度量单位（用户故事的数目）；
- 每星期结束时的总范围（3月22日那周是55个用户故事）；
- 成功交付的两个重要里程碑（里程碑1和里程碑2）；
- 每个星期完成故事的跟踪曲线（3月22日那周共完成了55个故事中的15个）；
- 该项目范围应该交付的最后期限（4月26日，第12个迭代结束时）。

基于易交流、可视化以及易维护性等方面的考虑，该图表应该由开发组长或迭代经理来管理，画在白板上，放在整个团队都非常容易看到的位置。

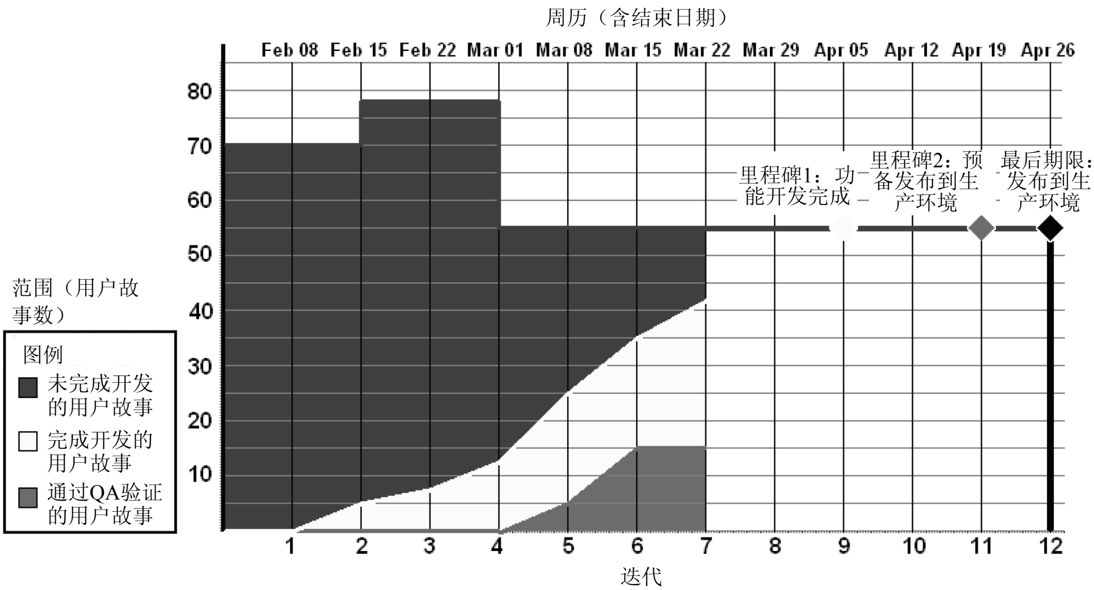


图8-1　项目范围增量图

定义范围的度量单位

为有了有效地得到项目范围增量，在定义范围的度量单位时，要取得所有项目成员的一致同意。而且这个度量单位会因具体项目不同而不同。在理想情况下，这一个度量单位在整个项目过程中不会变化。如果这个定义中途有变的话，历史数据就没有用了。

要避免使用小时数（或天数）作为度量单位。“度量范围”的意义在于了解还有多少内容需要完成，而不是还要多长时间能完成（即多少工作，而非多长时间）。如果使用时间的话，就成了估计时间与实际时间之间的联系，这会导致难于有效地度量并显示范围。

使用中期里程碑做为对照点来发现瓶颈

基于中期里程碑的进度比率会显示出交付流程运行是否良好。通过这一数值的对比可以发现交付瓶颈。比值的不同表明流程中有瓶颈存在。例如，当“功能完成”里程碑的比值大于“等待上线”里程碑时，表示QA环节是一个瓶颈。

再释“范围增量图”

对于示例中的范围增量图来说，该项目范围的度量单位为用户故事。项目开始之前，团队对

故事的度量单位做了定义，即一个故事如有下特征：

- 它表示一个或多个用例的全部或部分实现；
- 一个开发人员可以在2～5个工作日内实现它并完成单元测试；
- QA人员可以对其进行验收测试，以确保它满足需求。

在项目开始时，为项目范围设定了两个中期里程碑。里程碑1是开发所有用户故事并完成单元测试（但不一定是完成了QA工作）的时间点。里程碑2是完成所有用户故事、通过单元测试并完成QA工作的时间点。这个项目范围增量图直接反映出了完成中期里程碑所经历的过程。这也简明解释了该项目是如何管理范围的。

(1) 项目开始时，总范围只有70个故事。

(2) 在第二次迭代时，增加了8个故事，使总范围变成了78个故事。

(3) 在第四次迭代时，所有项目干系人在一起达成一致意见：根据该图表反映的历史趋势来看，团队不可以在预算内、以预期质量按时完成里程碑1和里程碑2。并一致同意削减项目范围。在后续会议中，所有项目利益关系人同意将23个用户故事推迟到下一个版本，因此，当前的项目范围变成了55个用户故事。

(4) 从第五次迭代开始，范围降为55个故事。

(5) 当前处于第八次迭代。范围没有变化，仍旧是55个用户故事。团队成员不能确定他们能否完成里程碑2，但他们决定不急于采取任何调整措施。

以下是用于产生范围增长图的原始数据。

迭代	范围	准备开发或正在开发	开发完成，等待QA验证	开发完成，但有严重bug	开发完成，通过QA验证
1	70	70	0	0	0
2	78	73	2	3	0
3	78	71	1	6	0
4	78	66	3	9	0
5	55	25	9	11	10
6	55	20	8	12	15
7	55	13	10	17	15

8.5 项目生命体征：交付质量

交付质量表示最终交付的产品的状况。该度量应该表明在项目范围内团队交付的质量有多好。

质量的信息指示器示例

缺陷数量图（图8-2）是用于度量和显示根据严重性分组后的系统缺陷数量。该图包括以下信息：

- 尚未解决的缺陷总数（到3月22日，即第七次迭代结束时，共计47个缺陷）；
- 在发布之前必须修复的缺陷总数（33个高优先级缺陷）；
- 可以推迟到下一个版本修复的缺陷总数（14个低优先级缺陷）；
- 每周的缺陷数（前两次迭代是0，第三次迭代是14个，第四次迭代是8个，第五次迭代是9个，第六次迭代是20个）。

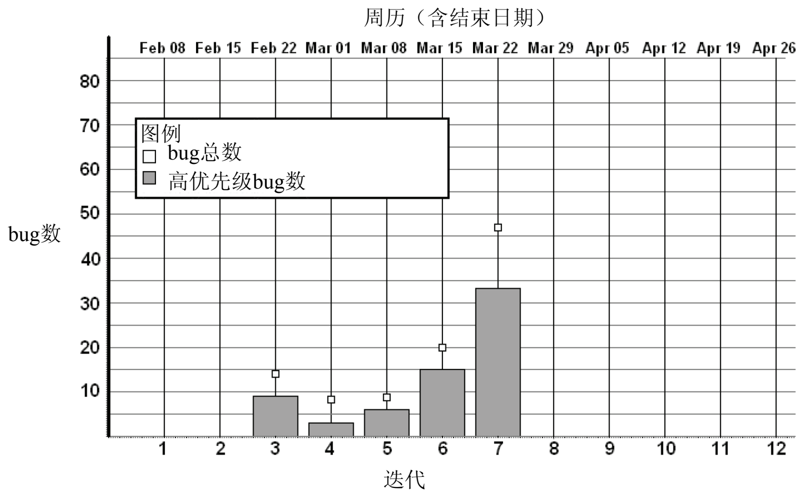


图8-2 bug数量图

为了方便交流、增强可见性以及易维护性，该图应该由测试人员维护，同样要在整个团队内可见。

再释缺陷数量图

当报告一个缺陷时，QA人员应指定其严重性（低、中、高级严重）。严重的缺陷是指那些阻碍测试，必须立即修复的缺陷。高优先级是指在发布到生产环境前必须修复的缺陷。中优先级是指最好能在发布到生产环境之前修复。低优先级是指能修复更好，但不用必须修复。

每周一的早上，QA人员会更新这个缺陷数量图。图中高优先级（High-priority）的缺陷是指那些严重性为“严重”和“高”的缺陷，低优先级（Low-priority）的缺陷是那些严重性为“中”

或“低”的缺陷。

在本例中，缺陷数量在前两个星期是零，因为QA团队还没有组建，没人进行验收测试。以下是生成图表的原始数据。

迭代的最后一天	严重bug数	高优先级bug数	中优先级bug数	低优先级bug数
1	0	0	0	0
2	0	0	0	0
3	0	9	4	1
4	0	3	4	1
5	0	6	2	1
6	0	15	3	2
7	3	30	10	4

8.6　项目生命体征：预算燃尽

预算燃尽代表根据范围交付情况所反应的预算状况。该维度表明了项目有多少预算，预算使用的比率，以及剩余预算还能维持多久。

预算的信息指示器示例

预算燃尽图是一种度量和显示已花费的项目预算，剩余预算，以及预算花费比率的方法（如图8-3所示）。该图表示如下信息：

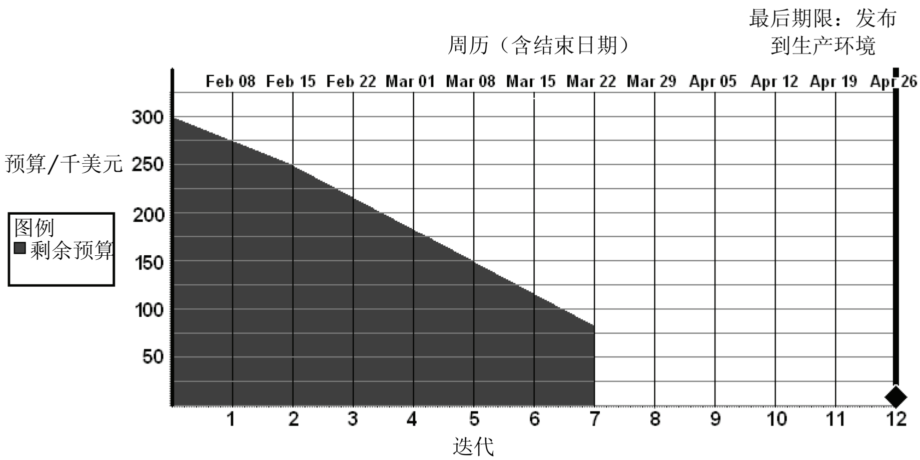


图8-3　预算燃尽图

- 预算的度量单位（千美元）；
- 总预算（项目开始时为300 000美元）；
- 当前预算花费额（到第七次迭代为止共计220 000美元）；
- 当前预算剩余（到第七次迭代为止共计80 000美元）；
- 每周的预算花费（前两次迭代为每周25 000美元，从第三次迭代开始，每周涨为33 333美元）；
- 最终交付日期（4月26日，第12次迭代结束以后）。

为了促进交流、增强可见性和易维护性，该图表应由项目经理维护更新，并为整个团队可见。

再释预算燃尽图

前两次迭代中，共有八个项目成员，每个人都租用一台开发用电脑，团队共享同一个构建及源代码服务器。每个月在这些团队成员及租用的电脑和服务器的成本为25 000美元。在第三次迭代时，两个新的项目成员、两台租来的开发用电脑以及一个测试服务器被加入到项目中。这些东西使每周的预算成本变成了33 333美元。

8.7　项目生命体征：当前开发状态

当前开发状态代表系统交付的实时状态。它表示在项目范围内每一项目交付的实时状态。

当前开发状态的信息指示器示例

用故事板（图8-4）以及故事卡（图8-5）来度量和显示当前系统的开发状态。这些图与卡显示出以下信息：

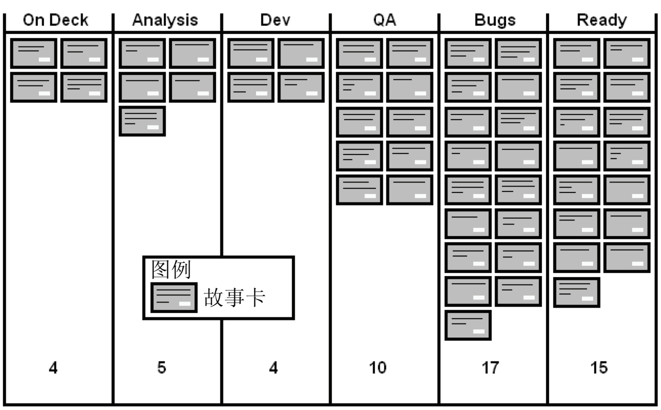


图8-4　故事板

- 每个故事卡都在项目范围之内（一共55个故事卡）；
- 每个故事卡可能的状态（On Deck，Analysis，Dev，QA，Bugs以及Ready）；
- 每个故事卡当前的状态（某一时间，每个故事卡只能有一种状态）；
- 每个状态下的故事卡数量（4个处于on Deck，5个处于Analysis，4个处于Dev，10个处于QA，17个处于Bugs，15个处于Ready）；
- 哪个人员当前正在处理哪个故事卡（浅色标签上是人员姓名，表示John正在处理35号故事）。

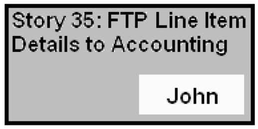


图8-5　故事卡

为了可见性、易沟通及易维护性，这个故事板应由分析人员、开发人员以及测试人员共同管理，并立于整个团队面前。

定义开发状态

对于每个项目来说，开发状态应该是唯一的。在图8-4中的状态并不一定适应于所有项目，也不必要适应于所有项目。对于状态定义，唯一的标准就是取得团队所有成员一致同意即可。

项目中期是可以改变这些开发状态的。而且在项目中期重新审视一下这些状态也是必要的，因为最原始的某些状态可能已经不适合于项目的当前情况了。

再释故事板与故事卡

故事板是第八次迭代的星期二下午3点14分的状态。下面是对每个状态的解释。

On Deck　故事的有待分析与实现
Analysis　故事正处于分析状态
Dev　开发并写单元测试
QA　故事已完成开发，并且完成单元测试，可以被QA人员检查了
Bugs　QA人员对故事进行了检查，并且发现了缺陷
Ready　QA人员对故事进行了检查，而且已发现的缺陷都被修复了

8.8　项目生命体征：团队感觉

团队感觉是团队对项目状态的感觉。该度量应该表明团队在项目交付的某个方面上的观点。

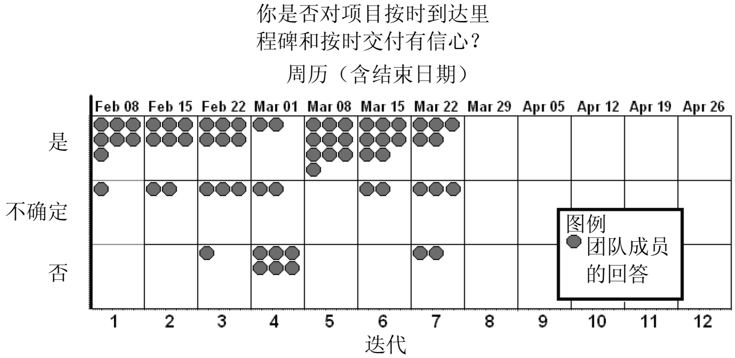


图8-6　团队心情图

团队心情的信息指示器示例

团队心情图（图8-6）用于度量和显示团队成员对项目状态感觉。该图包括以下信息：

- 每个星期在回顾会议上向团队成员提出的问题（“你对……有信心吗？”）；
- 团队成员可能的回答（“是的，” “不确定，”或“没有”）；
- 每个团队成员的回答（“当在第六个回顾会议时，十个人中有八个人回答“是的”）；
- 该图表应在每次回顾会议时由所有团队成员来更新，并让整个团队都能看到。

再释团队心情图

用圆点代表每个成员对问题的回答。收集回答时应使用匿名方式，以免互相影响，而且不应只有一个答案。

注：本项目中，项目成员数量发生了变化。前两次迭代，是八个成员，从第三次迭代开始，是十个成员。

第 9 章

消费者驱动契约：服务演化模式

Ian Robinson，架构师[①]

面向服务的架构（Service-oriented architectures，SOA）提升了组织的敏捷性，降低变化带来的整体开销。把高价值业务功能置于离散、可重用的业务之中，这样，连接和改编它们来满足核心业务过程就会容易一些。通过降低服务间的依赖性，变化带来的成本就会进一步降低，有助于快速重组和调整，响应未计划的事件或发生的变化。

业务可以完全享受到这些益处，前提是采纳SOA的服务能够彼此独立进化。获得这种独立性，一种常用的方式是，构建共享契约而非类型的服务。提供者发布契约，契约描述服务、发送和接收的消息、端点以及其采用的通信方式。随后，消费者实现契约，使用服务，而不必依赖于服务维护的内部领域表示，或服务基于的平台和技术。

在本文中，我会讨论服务契约以及其常见的实现和消费方式怎样带来服务的过度耦合。基于契约开发服务，会迫使我们在演化服务提供者时，也要以同样的速率演化消费者，因为服务的消费者会较为草率地用他们固有的逻辑来表述整个文档Schema，而将自己与提供者耦合在一起。减少耦合问题，有两个众所周知的策略：添加Schema扩展点和对接收的消息执行“刚好足够（just enough）”的校验。

服务最终彼此紧密耦合，其根源在于服务契约是以提供者为中心的。提供者契约时常导致某个消费者的期望和需求被遗忘，这是其本质所决定的。为了改变这种情况，我建议围绕消费者需求，也就是围绕消费者契约提供服务，消费者契约表示了消费者对服务所表示的业务功能拥有的合理期望。

如果服务引入了消费者契约，并在与消费者交换消息时遵循这个契约，那么它就实现了一种

① 准备这章时，很多人给予了我许多帮助，在此我想鸣谢：Ian Cartwright、Duncan Cragg、Martin Fowler、Robin Shorrock和Joe Walnes。

提供者契约的派生形式：消费者驱动契约。本文其他部分描述了一种基于断言的语言，利用这种语言，消费者驱动契约使提供者得以洞察消费者的义务，并让服务的演化围绕着交付消费者需要的关键业务功能来展开。

消费者驱动契约模式主要适用于可以在其中识别和影响消费者的服务社区，也就是适用于企业边界内的服务。这个模式有一些很明显的限制，比如，缺乏工具支持，它对消息处理管道的影响，以及由此给服务社区引入的日益增长的复杂度和协议依赖性等等。但是，我们相信，只要在合适的场景下应用这个模式，利就会远大于弊。尽管看起来它将服务间的通信变得更为复杂，但如果从寻求提升各种的细粒度洞察力和组织级敏捷所依赖的快速反馈来看，这个模式毫无疑问是敏捷的。服务间某种程度上的耦合是不可避免的，也是在预期之内的：消费者驱动契约将这种耦合变成对分析而言是已知、可以度量和接纳的。而且，这个模式在系统生命周期的开发、部署和操作等部分之间架起了一座桥梁，允许我们建立轻量级的版本策略并预估演化服务的影响和成本，这样的话，考虑到拥有一个系统的总成本，它对于履行我们的职责还是有所贡献的。

9.1 演化服务：一个例子

为了阐述演化服务时可能会遇到的一些问题，以一个简单的Product服务为例，该服务允许用户应用搜索产品名录。

这是一个搜索结果文档的例子。

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns="urn:example.com:productsearch:products">
  <Product>
        <CatalogueID>101</CatalogueID>
        <Name>Widget</Name>
        <Price>10.99</Price>
        <Manufacturer>Company A</Manufacturer>
        <InStock>Yes</InStock>
  </Product>
  <Product>
        <CatalogueID>300</CatalogueID>
        <Name>Fooble</Name>
        <Price>2.00</Price>
        <Manufacturer>Company B</Manufacturer>
        <InStock>No</InStock>
  </Product>
</Products>
```

Product服务目前由两个应用程序使用：一个内部的营销应用程序，一个外部分销商的Web应用程序。两个消费者都会在处理之前，先使用XSD校验接收到的文档。内部应用程序使用

CatalogueID、Name、Price和Manufacturer几个字段；外部应用程序使用CatalogueID、Name和Price几个字段。没有人用InStock字段。InStock是为市场应用所考虑的，在开发的早期就已经在那里了。

一种常见的演化服务的方式是：为满足一个或多个消费者的要求，给文档添加一个字段。但随着提供者和消费者实现方式的差异，即使这样简单的变化，都可能给企业及其合作伙伴带来昂贵的代价。

在这个例子中，Product服务成型一段时间后，又有一个分销商考虑使用它，但它要求为每个产品添加一个Description字段。按照消费者构建的方式，改变意味着昂贵的代价，对提供者和既有消费者都一样——变化的成本取决于变化是如何实现的。至少有两种方式在服务社区的成员间分摊改变的成本。首先，你可以修改原有的Schema，这需要每个消费者更新Schema，以便正确校验搜索结果；修改系统的成本由提供者和既有消费者分摊：提供者面临这样的请求总得做出些修改；但消费者可能对更新的功能并无兴趣。另一种方式，你可以选择向新用户开放另一个操作和Schema，保留既有消费者原有的操作和模式。这样，修改的成本就限制在提供者一方，但是服务会变得更加复杂，维护成本更加昂贵。

即便如此简单的例子都可以告诉我们，一旦提供者和消费者进入产品阶段，提供很快就会发现，修改提供给消费者的契约中的任何元素时都要谨慎。因为提供者既无法预期，也无法洞察消费者实现契约的方式。不去反省SOA中所实现契约的功能和角色，你就将服务置于一种隐藏的耦合之中，少有人能够以系统的方式对其进行阐述。对服务社区采纳契约的方式缺乏系统认知，对服务提供者和消费者选择的契约驱动实现方式缺乏足够约束，这两点动摇了传说中SOA给企业带来的益处。简而言之，你给企业带来了服务上的负担。

9.2 Schema 版本

接下来，我们会深入到契约和耦合问题之中，这是例子中那个Product服务的痛苦之源。我们从Schema版本问题出发。WC3技术架构组（The WC3 Technical Architecture Group，TAG）描述了若干Schema策略，有助于演化你服务的消息模式，这是一种缓和耦合问题的方式。[①]

两种极端都会带来问题：抑制业务价值的交付以及增加拥有系统的总成本。显式或隐式的“无版本”策略都可能让系统变得在交互上无法预期，变得脆弱，并会增加随后修改的成本。

另一方面，大爆炸策略会增加紧耦合的服务场景，随之而来的是Schema的变动影响到提供者

① 建议查看TAG Finding，“Versioning XML Languages [editorial draft]，”November 16, 2003；http://www.w3.org/2001/tag/doc/versioning。

和消费者、破坏正常的运行时间、延缓演化，并降低机会带来的回报。

例子中的服务社区有效地贯彻了大爆炸策略。考虑到提升系统业务价值相关的成本，很明显，提供者和消费者都会受益于一个更加灵活的提供向后/向前兼容的Schema版本策略（TAG Finding称其为兼容策略）。在讲到服务演化时，对于向后兼容的Schema，新Schema的消费者可以接收一个旧Schema的实例；向后兼容的服务提供者，能够处理新版本的请求，也能够接收根据旧Schema生成的请求。另一方面，对于向前兼容的Schema，旧Schema的消费者可以处理新Schema的实例。对于既有的Product消费者而言，这是一个关键点：如果在第一次部进入产品阶段时，搜索结果Schema做成了向前兼容，消费者就能够处理搜索结果的新版本，而无需任何修改。

扩展点

把Schema设计成向后/向前兼容是一项很好理解的设计任务，具有扩展性的Must Ignroe（必须忽略）模式很好的阐述了这一点[①]。Must Ignore模式建议：Schema由扩展点组成。这些扩展点允许我们向类型添加扩展元素，以及向每个元素添加额外的属性。这个模式还建议，XML语言定义一个处理模型，用以指出消费者如何处理扩展。最简单的模型要求消费者忽略它们无法识别的元素，这是这个模式得名的原因。最简单的模型也可能会要求消费者处理“必须理解”的标记，或是如果无法理解就中止元素的添加过程。

这是一个Schema，我们的搜索结果文档最初就是基于它的。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                      name="Product" type="Product" />
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
          <xs:element name="Name" type="xs:string" />
          <xs:element name="Price" type="xs:double" />
          <xs:element name="Manufacturer" type="xs:string" />
```

① David Orchard，“Extensibility，XML Vocabularies，and XML Schema”；http://www.pacificspirit.com/Authoring/Compatibility/Extending AndVersioningXMLLanguage.html。

```
        <xs:element name="InStock" type="xs:string" />
      </xs:sequence>
  </xs:complexType>
</xs:schema>
```

我们再来创建一个向前兼容的、可扩展的Schema：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
                        name="Product" type="Product" />
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
      <xs:sequence>
        <xs:element name="CatalogueID" type="xs:int" />
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Price" type="xs:double" />
        <xs:element name="Manufacturer" type="xs:string" />
        <xs:element name="InStock" type="xs:string" />
        <xs:element minOccurs="0" maxOccurs="1"
                        name="Extension" type="Extension" />
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Extension">
      <xs:sequence>
        <xs:any minOccurs="1" maxOccurs="unbounded"
                    namespace="##targetNamespace" processContents="lax" />
      </xs:sequence>
  </xs:complexType>
</xs:schema>
```

这个Schema在每个产品的末尾都包含了一个可选的`Extension`元素。这个`Extension`元素本身可以包含源自目标命名空间的一个或多个元素。

现在，当有人要求你为每个产品添加一个描述，你可以发布一个新的Schema，包含新增的*`Description`*元素，提供者将它插入到扩展容器中。这让Product服务可以返回包含产品描述的结果，也让使用新Schema的消费者可以校验整个文档。

使用旧Schema的消费者，即便它们并不处理这个描述，也不会有什么问题。新的结果文档是这样的。

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns="urn:example.com:productsearch:products">
```

```
  <Product>
        <CatalogueID>101</CatalogueID>
        <Name>Widget</Name>
        <Price>10.99</Price>
        <Manufacturer>Company A</Manufacturer>
        <InStock>Yes</InStock>
        <Extension>
          <Description>Our top of the range widget</Description>
        </Extension>
  </Product>
  <Product>
        <CatalogueID>300</CatalogueID>
        <Name>Fooble</Name>
        <Price>2.00</Price>
        <Manufacturer>Company B</Manufacturer>
        <InStock>No</InStock>
        <Extension>
          <Description>Our bargain fooble</Description>
        </Extension>
  </Product>
</Products>
```

修订过的Schema是这样的。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                      name="Product" type="Product" />
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
          <xs:element name="Name" type="xs:string" />
          <xs:element name="Price" type="xs:double" />
          <xs:element name="Manufacturer" type="xs:string" />
          <xs:element name="InStock" type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="1"
                      name="Extension" type="Extension" />
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Extension">
        <xs:sequence>
          <xs:any minOccurs="1" maxOccurs="unbounded"
                  namespace="##targetNamespace"
                  processContents="lax" />
        </xs:sequence>
  </xs:complexType>
```

```
  <code:bold><xs:element name="Description"
                             type="xs:string" /></code:bold>
</xs:schema>
```

注意，可扩展Schema的第一个版本对第二个向前兼容，第二个向后兼容第一个。然而，这种灵活性却是以增加复杂性为代价的。使用可扩展Schema，你可以对XML做出一些意料之外的变化；但是这也预示着它们可能是在为永远不会出现的情况做准备。这种做法在领域语言中引入了元信息容器元素，干扰了简单设计所带来的表达力，也破坏了对业务信息有意义的表述。

这里，我就不进一步讨论Schema的可扩展性。这足够说明，扩展点允许你对Schema和文档做出向后/向前兼容的变化，而不对服务提供者和消费者造成破坏。然而，当你对契约做出明显的破坏性变化时，Schema扩展就帮不上你了。

9.3　破坏式的变化

作为一种增值，Product服务在搜索结果中包含了一个字段，表示该产品是否有库存。为了得到这个字段，这个服务会对一个遗留的清单系统进行一次代价高昂的调用。这是一种维护成本极高的依赖。服务提供者希望移除这个依赖、清理设计并改善系统的整体性能，同时最好不强加给消费者任何修改成本。幸运的是，没有消费者应用程序会实际用到这个值；虽然昂贵，却是冗余。

这是个好消息。坏消息是，按照我们目前的设置，如果要从我们的扩展Schema移出一个必要的组件（这个例子里的InStock字段），就会破坏既有消费者。为了修复提供者，就不得不修复整个系统。当我们从提供者中移除这个功能，发布新的契约，每个消费者应用都不得不采用新Schema重新部署。我们也不得不对服务间的交互进行一次全面的测试。这样一来，产品服务无法独立于其消费者进行演化：提供者和消费者必须同步变化。

9

这个例子中的服务社区演化受阻，因为每个消费者都实现了一种“隐藏”的耦合，这种耦合会将整个提供者契约反映在消费者内部逻辑中。消费者使用XSD校验和源自文档Schema的静态语言绑定，隐式地接受了整个提供者契约，尽管它们想处理的可能只是部分组件。

David Orchard提供了一些线索，也许可以帮你避免这个问题，他提到了因特网协议健壮性原则：“总体说来，实现必须有着保守的发送行为和自由的接受行为。”

在提及服务演化时，我们可以放大这个原则，消息接收者应该实现“刚好足够”的校验；也就是说，它们应该只处理对实现业务功能有作用的数据，应该以受限的方式校验接收的数据，而非采用隐式的、不受限的、XSD处理中固有的“全有或全无（all-or-nothing）”校验。

Schematron

一种改进消费者端校验的方式是，沿着所接收消息文档树的轴进行模式表达式断言，其中可能使用结构化树模式校验语言，比如Schematron。[①]使用Schematron，Product服务的每个消费者就可以编程对搜索结果中预期发现的内容进行断言。

```
<?xml version="1.0" encoding="utf-8" ?>
<schema xmlns="http://www.ascc.net/xml/schematron">

  <title>ProductSearch</title>
  <ns uri="urn:example.com:productsearch:products" prefix="p"/>

  <pattern name="Validate search results">
        <rule context="*//p:Product">
          <assert test="p:CatalogueID">Must contain
                                        CatalogueID node</assert>
          <assert test="p:Name">Must contain Name node</assert>
          <assert test="p:Price">Must contain Price node</assert>
        </rule>
  </pattern>

</schema>
```

Schematron实现通常将一个Schematron Schema转化为一个XSLT变换，消息接收者可以用这个XSLT变换确定文档的有效性。

注意，在这个例子里面，对消费应用程序不处理的元素，Schematron Schema并不进行断言。按照这种方式，校验语言的目标限定为一组必要元素。只要对文档Schema的修改不破坏Schematron Schema显式描述的预期，就不会在校验过程起作用，即便修改会涉及删除那些曾经必备的元素。

对于契约和耦合问题来说，有这样一个相对轻量级的解决方案：不需要在文档中添加难于理解的元信息元素。接下来，我们再一次回退时间，将Schema恢复到本章开始的简单状态。但是这一次，我们要强调，消费者的接收行为是自由的。这意味着，它们应该只校验和处理支撑其业务功能的信息（使用Schematron Schema，而非XSD校验接收的信息）。现在，如果要提供者为每个产品添加一个描述，服务可以在不破坏既有消费者的情况下，发布修订后的Schema。类似的，如果发现没有任何消费者校验或处理InStock字段，服务就可以修订搜索结果Schema，而不会打乱每个消费者演化的速率。

在这个过程的最后，Product会让Schema变成这样。

---

① Dare Obasanjo，“Designing Extensible, Versionable XML Formats”；http://msdn.microsoft.com/library/en-us/dnexxml/html/ xml07212004.asp。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                          name="Product" type="Product" />
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
          <xs:element name="Name" type="xs:string" />
          <xs:element name="Price" type="xs:double" />
          <xs:element name="Manufacturer" type="xs:string" />
          <xs:element name="Description" type="xs:string" />
        </xs:sequence>
  </xs:complexType>
</xs:schema>
```

9.4　消费者驱动契约

在前面的例子中，对于提供者和消费者之间的契约，使用Schematron带来了一些有趣的见解，其内涵超越了文档验证。本节概括了一些见解，并将其表述为消费者驱动契约模式。

首先要提及的是，文档Schema只是服务提供者向消费者提供的内容的一部分。你可以把一个服务所有外部可利用点的总和称为提供者契约。

提供者契约

提供者契约用一个支撑业务功能必需的可导出元素集合，将服务提供者的能力表述出来。从服务演化的角度来看，契约是一个容器，包含了一套可导出业务功能元素。对于这些元素，下面是一个非规范的列表。

- 文档Schema：我们已经比较详尽的讨论过文档Schema。除了接口之外，文档Schema是提供者契约中最有可能随业务演化而变化的一部分；但是也许正因为如此，这也是我们拥有最多经验的部分，这种经验渗透在诸如扩展点和文档树路径断言之类的服务演化策略之中。
- 接口：在最简单的形式中，服务提供者接口包含了一套可导出操作的签名，消费者可以利用它们驱动提供者的行为。面向消息系统通常会导出一些相对简单的操作签名，将业

务信息变成交换的消息。在面向消息系统中，根据消息头或负载所承载的语义，接收到的消息驱动着端点的行为。另一方面，RPC类服务更多的是把业务语义编码在操作签名中。每种方式，消费者都依靠提供者接口的一部分来实现业务价值，因此，演化我们的服务场景时，我们必须考虑接口的消费方式。

- 会话：服务提供者和消费者在会话中交换消息，会话由一个或多个消息交换模式组成。会话过程中，消费者可以预期提供者所发送和接受的消息，这样，交互特有的一些状态就会显现出来。比如说，宾馆预定服务也许会为消费者提供这样的能力：在会话之初预约房间，在随后的消息中确认预订并收取订金。这里的消费者可以合理地预期服务在随后的信息交换中“记住”预约的细节，而不是在过程中的每一步重复整个会话。随着服务的演化，提供者和消费者之间会话方式可能会发生变化。因此，会话就应该纳入提供者契约考虑的范畴。
- 策略：除了导出文档Schema、接口和会话之外，服务提供者也许还要声明和强制执行特定的使用需求，用以管理契约的其他元素如何使用。最常见的是，这些需求同安全和事务上下文相关，这个上下文主要是关于消费者如何利用提供者的功能。Web服务栈表示这种策略框架，通常是用WS-Policy通用模型，加上领域特定的策略语言，比如WS-SecurityPolicy，但是，我们将策略纳入提供者契约进行考虑，在这样的上下文中，我们的策略定义就不依赖于具体的规范和实现。[①]
- 服务特征的质量：评估业务提供者和消费者所用业务的价值潜力，通常是要在特定服务特征质量的上下文中进行，比如可用性、延迟和吞吐量。你应该把这些特征看作提供者契约的组成部分，在你的服务演化策略中予以考虑。

对比谈及服务时通常所听到的定义，这里的契约定义更加宽泛，但是，从服务演化的角度而言，这个定义是一种抽象，抽象了对问题领域有影响的重要力量。这说明，考虑到提供者契约所包含元素的种类，这个定义并不详尽；它只涉及一个可导出业务功能元素的逻辑集合，这些元素是一些候选元素，包含在服务演化策略中。从逻辑的角度而言，这套候选元素是开放的，但是实际上，一些内外部因素，比如互操作性需求或平台限制，可能约束着契约所包含元素的类型。比如，符合WS-Basic Profile的服务，属于它的契约不可能包含策略元素。

尽管有如此约束，契约的作用域还是可以由其成员元素的内聚性确定。契约可以包含许多元素，作用域可以很宽广，也可以只关注于一些部分，只要能够表达出业务功能的能力。

如何确定是否要在提供者契约中包含候选的契约元素呢？你可以问问自己的任何一个客户，他们是否都会合理地预期，在服务整个生命周期中，该元素的业务功能能力都一直都要得到满足。

① “Schematron: A Language for Making Assertions About Patterns Found in XML Documents”；http://www.schematron.com。

你已经看到了，在例子中，服务消费者会如何表达其对服务导出文档Schema的兴趣，以及如何断言，以便它们对契约元素预期能够继续得到满足。因此，文档Schema是提供者契约的一部分。

提供者契约有如下特征。

- 封闭而完整：根据消费者可用的导出元素全集，提供者契约表述了服务的业务功能能力，对于系统可用的功能而言，这个集合本身是封闭而完整的。
- 单一而权威：对于表达系统可用的业务功能而言，提供者契约是单一而权威的。
- 受限的稳定性和不变性：对于受限的时间和地点[①]而言，提供者契约是稳定而不变的。提供者契约通常用某种形式的版本区分受到不同限定的契约实例。

消费者契约

如果你决定考虑消费者对公开Schema的期望——认为提供者值得了解它们——就需要将那些消费者期望导入到提供者中。在这个例子里面，Schematron断言看上去非常像某种测试，对于提供者而言，这有助于确保提供者持续满足对客户的承诺。通过实现这些测试，提供者可以更好的理解，如何在不破坏服务社区的情况下演化消息结构。如果所做的修改实际上对一个或多个消费者造成了破坏，提供者就可以立即了解到这个问题，因此最好将相关的各方都考虑进来，在业务因素有要求时，迁就他们的需求或是鼓励他们进行改变。

在这个例子中，你可以说，所有消费者生成断言的集合表达了消息的必备结构，对它们的父应用程序，该消息就是在断言一直有效的阶段所交换的消息。如果将这套断言给予提供者，它就可以确保所发送的每条消息对每个消费者而言都是有效的，但是，前提是这套断言是有效而完整的。

将这个结构泛化，你可以将提供者契约同单独的契约义务区分开来，契约义务是特定于提供者-消费者关系的实例，现在，我希望称这种关系为消费者契约。如果提供者接受和采纳消费者表达的合理期望，它就是一种消费者契约。

消费者契约有如下特征。

- 开放而不完整：考虑到对系统可用的业务功能，消费者契约是开放而不完整的。从消费者对提供者预期的角度来看，它们表述的是系统业务功能能力的一个子集。
- 多个而非权威：按照服务的消费者数量的比例，消费者契约是有多个的。然而，对于提供者制订的契约义务全集而言，每个消费者契约都是非权威的。从消费者扩展到提供者，

① WS-Policy；http://www-128.ibm.com/developerworks/library/specification/ws-polfram。

这种关系的非权威属性是区分面向服务架构和分布式应用程序架构的关键特征之一。服务消费者必须认识到，它们在服务社区中对应物应该可以按照与自己完全不同的方式来消费提供者。对应物的演化，可能是以不同的速率和不同的需求变动进行的，这种变动可能会破坏存在于系统其他部分的依赖或预期。消费者无法预期对应物会如何或在何时破坏提供者契约；而分布式应用程序的客户端没有这种顾虑。

- 受限的稳定性和不变性：类似于提供者契约，消费者契约是在特定的时间段和 / 或位置有效的。

消费者驱动契约

消费者契约允许你在提供者生命周期内的任意点反映可以利用的业务价值。通过表述和断言提供者契约的预期，消费者契约被有效地定义了：在系统所实现的业务价值中，提供者契约当前支持哪部分，不支持哪部分。这让我给出这样的建议，服务社区也许会从"按照消费者契约的方式，在第一个实例中指定"的方式受益。在这种观点中，提供者契约满足了消费者的预期和需求。为了反映这种新契约组织派生出来的属性，可以称这种提供者契约为消费者驱动契约或派生契约。

消费者驱动提供者契约的派生属性为服务提供者和消费者之间的关系中添加了他治的一面。也就是说，提供者会受到源自其自身边界之外义务的影响。这决不会影响它们的实现所具备的基本的自治属性；它揭露了一个事实，成功的服务依赖于如何消费它们。消费者驱动契约有如下特征。

- 封闭而完整：从既有消费者需要功能的完整集合来看，消费者驱动契约是封闭而完整的。契约表述了在一定阶段可导出元素的必备集合，这些元素是支撑消费者预期所必需的，对它们的父应用程序而言，这个阶段是那些预期一直有效的阶段。
- 单一而非权威：以对系统可用业务功能的表述而言，消费者驱动契约是单一的，但是也是非权威的，因为它们派生自既有消费者预期的联合。
- 受限的稳定性和不变性：对于消费者契约的一个特定集合而言，消费者驱动契约是稳定而不变的。这就是说，根据消费者契约的一个特定集合，你就可以确定消费者驱动契约的有效性，在时间和空间上，有效地绑定了契约向前和向后兼容属性。对于特定的消费者契约和预期的集合而言，契约的兼容性保持稳定而不变，但是也易于随预期变化而变。

契约特征总结

下面的表格总结了本章描述的三种类型契约的特征。

契　　约	开放性	完整性	数　　量	权威性	受限于
提供者	封闭	完整	单一	权威	空间 / 时间
消费者	开放	不完整	多样	非权威	空间 / 时间
消费者驱动	封闭	完整	单一	非权威	消费者

实现

消费者驱动契约模式推荐使用消费者和消费者驱动契约构建服务社区。然而，这个模式并不指定消费者和消费者驱动契约所应采纳的形式或结构，它并不决定消费者预期如何同提供者交流，以及如何在提供者生命周期内进行断言。

契约可能有多种方式进行表达和组织。在最简单的形式中，可以用电子表格或类似的文档捕获消费者预期，在提供者应用程序的设计、开发和测试阶段予以实现。再继续向前一点，引入单元测试，对预期进行断言，你就可以确保契约在每次构建中都是以可重复、自动的方式描述和坚持下来的。在一些久经考验的实现中，预期描述为Schematron，或是类WS-Policy的断言，运行时在服务端点的输入输出管道中进行处理。

如同契约结构的情况一样，谈及在提供者和消费者间交流预期，可以有几个选择。既然消费者驱动契约模式是不了解实现的，考虑到恰当的组织设置，你可以通过和其他团队交流，或使用email传递预期。当预期和 / 或消费者的数量增长得大到无法以这种方式管理时，你可以考虑将一种契约服务接口和实现引入到服务基础设施中来。无论机制如何，交流很可能就变成了在带外（out-of-band）进行，而且会在任何运用系统业务功能的会话之前进行。

益处

谈及演化服务，消费者契约提供了两个明显的益处。首先，其关注于关键业务价值驱动的服务功能的规范和交付。服务只在其消费的程度上对于业务而言是具备价值的。消费者驱动契约将服务演化绑定于业务价值上，其做法是断言可导出服务社区元素的价值，也就是说，消费者向提供者要求用以完成其工作的东西。其结果是，提供者导出一个精益的契约，该契约明确地满足了支撑其消费者的业务目标。服务演化出现了，在这里，通过修改消费者对提供者的预期，它们表达出一个清晰的业务需要。

当然，从最小需求集合出发，演化服务以满足消费者预期的变化，这种能力假定你能够以一种可控和高效的方式演化和操作服务。因为它们并不围绕服务消费捕获任何预期，提供者契约必须补充一些其他机制，以监控和评估变化的影响。另一方面，消费者契约会给提供者灌输一套知

识，一个你可以在系统生命周期的操作部分期间所利用的反馈机制。使用派生自消费者驱动契约的细粒度观察和快速反馈，你可以计划变化，评估它们对当前处于产品阶段的应用带来的影响。实践中，这让你可以定位于个体消费者，鼓励其放弃某个预期，以避免其阻止你进行与当前并不向后和 / 或向前兼容的变化。

消费者驱动契约和SLA

我们已经讨论了消费者和消费者驱动契约表达业务价值的方式。但是，我应该澄清一下，尽管同WS-Agreement和Web Service Level Agreement（WSLA）这样的规范有一些表面的相似，但消费者驱动契约的目的却不是要表达服务层次上的协定。[①]WS-Agreement和WSLA是由需要带动的，这种需要是为消费者提供了一种保证，保证服务质量和资源的可用性，在WSLA的情况下，还有动态提供服务和分配资源的需求。在消费者驱动契约模式背后，有个潜在的假设，服务本身对业务是没有价值的；它们的价值在于被消费。按照消费者实际如何使用服务指定服务，我们的目标是交付组织的敏捷，以某种允许可控的服务演化的方式利用业务价值。

也就是说，WS-Agreement和WSLA都是充当例子的角色，表现出自动化契约协议和基础架构的样子。这两个规范都描述了协定模版，模版可以由任何表示和监控协定条件的断言语言组成。协定通过Web服务接口建立，这种接口完全保持了服务的独立，可以通过向服务管道中注入处理器进行监控。

义务

我们已经识别出将消费者驱动契约引入到服务场景的动机，描述了消费者驱动契约模式如何表示确定服务演化的力量。在本文结尾，我们要讨论一下这种模式的适用范围，以及实现消费者和消费者驱动契约时可能会遇到的一些问题。

消费者契约模式适用于这样的上下文，单一企业或是拥有良好服务的封闭社区，或是更特定一些，某种令提供者可以对消费者如何同它们建立契约施加影响的环境。[②]无论交流和表示预期和义务的机制如何轻量级，提供者和消费者必须了解、接受和采纳一套通道和约定。这无可避免的增加了一个复杂度的层次，以及对已经很复杂的服务基础结构的协议依赖。对于描述、实现和操作契约，缺乏工具和执行环境的支持，会导致这个问题进一步恶化。

---

① 参见http://www-128.ibm.com/developerworks/webservices/library/specification/ws-secpol。

② 参见Pat Helland的文章“Data on the Outside vs. Data on the Inside: An Examination of the Impact of Service Oriented Architectures on Data”中“Validity of Data in Bounded Space and Time”一节。http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp。

我建议过，围绕消费者驱动契约构建的系统能够更好的管理对契约的破坏性变化。但是，我并没有说这个模式就是解决破坏性变化问题的万能药；说一千，道一万，破坏性变化依旧是破坏性变化。然而，我确实相信，这个模式让我们更清楚地认识破坏性变化是怎样构成的。简而言之，破坏性变化就是不能满足既有消费者预期的东西。这个模式有助于识别破坏性变化，可以充当起服务版本化策略的基础。然而，正如前面已经讨论过的，实现了这个模式的服务社区最好能够在破坏性变化影响系统健康之前，先预期服务演化的影响，并识别潜在的破坏性变化。尤其是让开发和操作团队可以更有效地计划它们的演化策略——也许是反对某一特定阶段的契约元素，同时，对于持反对意见的消费者，鼓励它们升级至新版本的契约。

消费者驱动契约不见得会降低服务间的耦合。Schema扩展和“刚好足够”校验也许有助于降低服务提供者和消费者间的耦合，但即便如此，松耦合服务也会有一定程度的耦合。尽管不能直接减少服务耦合，但消费者驱动契约确实可以将一些残留的“隐藏”耦合挖掘出来，以便提供者和消费者可以更好地处理和管理这些耦合。

结论

SOA带来了组织的敏捷，降低改变成本，但前提是服务能够彼此独立地演化。消费者草率地实现了提供者契约，这样的方式会引起过度耦合的服务。Schema扩展点的Must Ignore模式，以及使用Schematron断言实现的“刚好足够”的模式校验策略会让消费者受益，这么做降低了消费者本身与提供者之间的耦合。另一方面，提供者与消费者之间有一套用以交流的契约，服务提供者会对从这套契约派生出来的契约的运行时职责得到更多的观察和反馈；消费者驱动契约对服务演化的支持贯穿于服务的整个运作生命周期，而且更贴近于规范以及具有关键业务目标的服务功能的交付。

第 10 章

领域标注

10

Erik Doernenburg，技术负责人

10.1 当领域驱动设计遇到标注

在过去十年里，众多软件开发者意识到：软件应用程序中最大的复杂度来自于软件要处理的实际问题领域。正因为如此，所谓“领域驱动设计”有两个前提：

- 大部分软件项目关注的焦点是业务领域和领域逻辑；
- 复杂的领域设计应该基于模型来进行。

换句话说，领域驱动设计将用业务领域术语表述的业务领域的面向对象模型置于软件系统的核心。数据通常保存在关系数据库里，但看待数据的主视角是领域对象，而不是数据库表和存储过程。核心业务逻辑集中保存在领域模型中，而不是散布在用户界面和应用程序的服务层里。

如果遵循领域驱动设计方法，得到的软件系统就会清晰地区分出领域模型和基础设施（及界面）：前者通常较为稳定、生命周期较长；后者通常生命周期较短，并且与具体的技术（例如O/R映射或者web框架）相关。这里的挑战在于如何保持两者之间的分野，从而使两者能够各自保持可复用性：一方面，领域模型应该能够在不同的应用程序、不同的服务中使用，或是在技术升级之后继续使用；另一方面，基础设施应该能用于支撑各种领域模型。当然最终的目标是实现基础设施的行业标准化（不论采用商业软件还是开源软件），从而使应用程序开发者能专注于问题领域。

领域特有的元数据

理想情况下，应用程序应该基于领域模型来组装，并利用自动化手段来生成最终的软件，这样领域模型和基础设施都不需要代码上的改变。很自然地，这需要大量使用反射和泛型类型，不

过与此同时，基础设施代码也将受惠于领域的元信息。

大量元数据是作为模型的实现而呈现的。譬如说，"'部门'与'员工'存在一对多关系"这一事实是以"部门"类与"员工"类之间的关系来呈现的。我们在"部门"类里使用一个集合，其中包含多个"员工"对象，这就是在告诉其他代码：这里有一个"对多"的关系；而在"员工"类里有一个"部门"对象的引用，于是两者之间的关系就可以确定为"一对多"。基于这些信息，用户界面框架就可以选择适当的窗体控件（例如用下拉列表框来列出所有员工）来为一个部门生成界面。

隐式的元数据——也就是作为实现的一部分而呈现出来的元数据——已经能支持很大程度的自动化；但如果将更多的元数据显式地呈现出来，对应用程序将大有好处。这种好处尤其明显地体现在数据校验上。在前面这个"部门与员工"的例子中，界面框架虽然可以自动生成"员工列表维护"的界面，却无从知晓"没有任何员工的部门"是否合法。如果给"员工"集合加上元数据来说明这是一个"1..n"的关系，那么界面框架就可以阻止用户保存"没有任何员工的部门"。这样的信息固然可以用特别定制的集合类来描述，不过现代的开发平台提供了一种更适合此用途的语言结构。

Java的标注和.NET的特性

编程语言的设计者们使元数据具有提供抽象和降低耦合度的能力。从第一个发行版本开始，微软的.NET平台及其通用语言运行时（CLR）就提供了一种机制，让程序员能够为几乎任何语言元素创建和添加任意的元数据。CLR把这个概念称为特性（attribute）。特性的定义与其他类型一样，它也有自己的数据和行为；但使用特性的语法有些特别：需要用方括号将其作用于其他语言元素。

下面的例子展示了C#中的一个特性，其用途是指定并校验一个属性（property）值的最大长度。

```
[AttributeUsage(AttributeTargets.Property)]
public class MaxLengthAttribute : Attribute
{
    private int maxLength;

    public MaxLengthAttribute(int maxLength)
    {
        this.maxLength = maxLength;
    }

    public void validate(PropertyInfo property, object obj)
    {
```

```
        MethodInfo method  = property.GetGetMethod(false);
        string propertyValue = (string)method.Invoke(obj, new object[0]);
        if(propertyValue.Length > maxLength)
        throw new ValidationException( ... );
    }
}
```

除了继承自Attribute类之外，用来定义这个特性的MaxLengthAttribute类本身还被附着了一个AttributeUsage特性。AttributeUsage特性是由CLR定义的，它在这里指定MaxLengthAttribute特性只能作用于属性，而不能作用于别的语言元素（例如类或者成员变量）。MaxLengthAttribute的实现假设其所作用的属性是string类型的，但这层约束条件并不能作为特性定义的一部分得以表达。从validate方法中可以看到，在特性定义中经常使用反射来达到解耦。值得一提的是，validate方法必须由应用程序代码来调用触发，CLR没有提供“当目标属性被访问时自动调用特性中的代码”的机制。

下面的例子展示了如何在代码中使用这个特性。

```
[MaxLength(50)]
public string Name
{
    get { return name; }
    set { name = value; }
}
```

从Java 5开始，Java也开始提供类似的语言结构。在Java的世界里，这种语言结构叫做标注（annotation）。我认为在软件开发的语境之下，这个名称更不易混淆，因此更好地描述了它所指代的概念。为此，在本文剩下的部分中，我将用标注这个词来指代与此概念相对应的各种实现方式。

Java也使用了一种以“@”字符开头的特殊语法来使标注作用于其他语言元素。但与.NET的版本相比，Java的实现有几处显著不同。首先，在用Java定义一个标注时，我们并不使用继承，而是使用一个新的关键词@interface。其次，在Java中也不指定该标注能被作用于什么语言元素，而需要指定它作用于开发周期的哪些阶段。不过最大的差异还是在于：Java的标注不包含代码，它们更像是接口而不太像是类。所以，.NET的版本可以用构造函数来提供默认值，而Java标注则必须用特别定义的default关键字。

因为有此差异，结果是显而易见的：对Java标注而言，数据校验行为必须在另一个类中实现。这本身并不是一个大问题，但确实有悖于封装原则。.NET的版本可以保持最大长度的值为私有值，因为标注本身就包含了校验逻辑；而Java的版本则必须将最大长度的值定义为公有值，并将标注传递给位于另一个类的校验方法。这里呈现出“依恋情结”（feature envy）的代码坏味道，并且会导致平行的类体系。不过应该指出，大部分情况下我们使用标注时并不需要它们包含类似

这样的逻辑，因此大部分情况下Java标注与.NET特性的用法是一致的。

领域标注

如果给标注起一个合适的名字，用它来表达领域相关的信息，我们就叫它领域标注（domain annotation）。

领域标注有三大特点：

- 它们只作用于领域对象中的语言元素，例如领域类、领域特有的公有方法或是这些方法的参数等。
- 它们是领域模型的一部分，与领域对象位于同样的包/命名空间中。
- 它们提供的信息可以在应用程序的多个功能中使用。

第一大特点是毋庸置疑的：如果一个标注作用于领域模型之外的对象，那么它当然就不能算是领域模型的一部分。另外，尽管有时环境会要求领域对象实现某些特定的方法（简单如`equals`、`hashCode`等，复杂如序列化逻辑），但领域标注通常不用于实现这些方法。类似地，私有方法明显是对象的内部实现细节，因此也不需要用标注来提供更多关于它们的信息。

当然这条规则也有例外：前面的“一对多关系”的例子中所需要的元数据在各种问题领域中都可以使用，所以EJB3标准就提供了通用的`@OneToMany`标注。但这样的标注不是定义在领域模型中，而是定义在基础设施框架中，这就违背了我们的第二条规则，因此使用这一标注就会使领域模型与EJB3规范相耦合。更糟糕的是，如果应用程序的基础设施代码也需要这部分信息，那么它也会与EJB3规范相耦合。显然，此刻在“保持领域模型不依赖于基础设施代码”与“以通用的基础设施提供可复用的标注”之间存在着冲突。和往常一样，这里并没有一个明确的答案告诉我们应该选择哪一边。

领域标注的例子

前面我曾经提到：标注主要用于提供数据。以CLR特性而言，它们永远不会出现在被标注元素的执行路径上。也就是说，必须由额外的代码来使用这些标注。尽管有这些额外的代码存在，使用标注还是能减少总体代码量，使实现更加清晰。考虑用另一种方法来实现最大长度校验的例子：可以给领域对象加上一个校验方法（依据命名惯例，或是明确添加一个接口），其中的代码大致如下。

```
public void validate()
{
    if(getName().length > 50)
```

```
        throw new ValidationException("Name must be 50 characters or less");
    if(getAddressLineOne().length > 60)
        throw new ValidationException(
                        "Address line one must be 60 characters or less");
    /* more validation of the object's state omitted */
}
```

这个方法显然很有问题，因为它将几方面的关注点混合在一起。假如稍后系统需要同时报告各项校验错误，这段代码就无法胜任。所以，大部分程序员会抽取一个方法用于记录校验错误，于是代码就会变成这样。

```
public void validate()
{
    if(getName().length > 50)
        validationError("Name", 50);
    if(getAddressLineOne().length > 60)
        validationError("AddressLineOne", 60);
    /* more validation of the object's state omitted */
}
```

将抽象层次再提高一步，可以创建一个单独的校验方法，将要校验的字段和最大长度传递给它，它就可以完成校验工作。该方法的实现会用到反射。于是`validate`方法中的代码就会变成这样。

```
public void validate()
{
    validate("name", 50);
    validate("addressLineOne", 60);
}
```

这样一来，所有能抽取出来的代码都被移到了一个单独的方法中，`validate`方法中剩下的就是一系列元数据。不过，在标注的帮助下，代码还可以变得更好。

```
@MaxLength(50)
public String getName()
{
    /* implementation omitted */
}
```

使用标注不仅让“最大长度”的信息紧挨着`getName`方法出现，而更重要的是避免了用字符串来指代方法。与前一个版本相比，引入标注之后唯一增加的代码就是遍历所有方法并找出其中有`MaxLength`标注的那些。

10.2　案例分析：Leroy 的卡车

设计模式并不是由什么天才凭空发明的，而是从现有代码中观察和重构出来的。领域标注也

是一样。本节将介绍的两个领域标注与前面所介绍的`MaxLength`标注颇有异曲同工之妙。（由于这个例子出自ThoughtWorks为客户开发的系统，出于商业原因，这里不会给出真实的标注或源代码。）

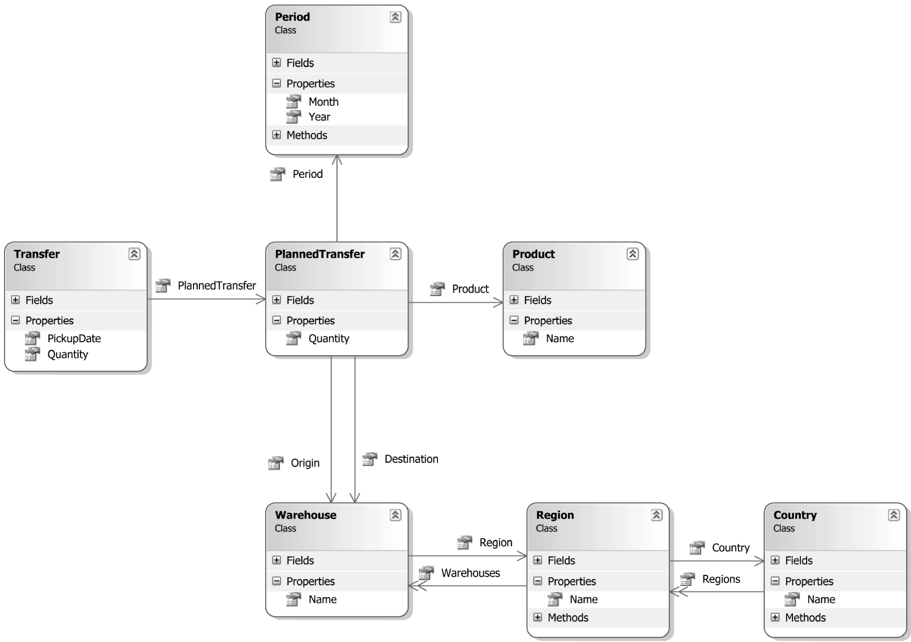


图10-1　运输事务

“Leroy的卡车”是我和同事Mike Royle一起创造的用于展示如何使用领域标注的一个示例应用程序。它出自前面提到的ThoughtWorks项目的经验。和原来的项目一样，这里的问题领域是物流运输行业。应用程序本身是一个智能客户端——它是一个Windows应用程序，数据保存在服务器端，在断开网络连接的情况下客户端也可以工作。和原来的应用程序一样，这个示例应用程序以C#实现，但我们升级到了2.0版本，以使代码更干净。

领域模型

在这个案例分析中，我们主要关注两个领域模型：对仓库间货物运输的建模，以及对系统用户及其角色的建模。

货物运输模型的核心是`PlannedTransfer`领域对象，它代表了一次被纳入计划的运输：在指定的时间段（指定年份的某一个月）内，将一定数量的货物从来源仓库运到目标仓库。实际的运输行为则是由`Transfer`领域对象来表示的，该对象会引用对应的`PlannedTransfer`对象。一次运输还包含其他相关信息，例如起运日期（不一定要在计划的时间段内）和实际运输的货物数量等。

此外还有几个与地理相关的领域对象值得一提：作为运输的起点和终点，仓库一定位于某个地区，而后者又一定属于某个国家。

为了案例分析的便利，所有这些领域对象都已经被简化，只剩下了必要的属性用于展现领域标注的用法。原来那个真实项目的领域模型要复杂得多：代表产品的`Product`类就有7个属性；我们用了6个类来建模各种不同的运输类型，`Transfer`类还包含运输方式、合同等信息。所以，如果你想找出更好的方式来为这个案例分析中的问题领域建模，请一定记住：它是经过了大量简化的。

`User`领域对象代表了系统的使用者。在这个简化的模型中，每个用户有一个名字，并且与一个国家（也就是该用户所在的国家）相关联。此外用户还可以有多种角色，例如计划制订者（Planner）、国家级管理员（Country Admin）、全球管理员（Global Admin）等。计划制订者可以创建和修改运输计划；国家级管理员负责维护其所在国家的仓库、地区和用户数据；全球管理员则可以往系统中新增国家并指定国家级管理员。

数据分类

刚开始尝试引入领域标注，我们就遇到了数据分类的问题。在前面对领域模型的介绍中，各种领域对象似乎没什么差别——它们都是描述某些领域概念的对象。但我们也曾提到，仓库、地区和国家是与地理相关的领域对象，说明它们之间确实有着某些特别的共性。此外从角色的介绍也能看出，具有不同角色的用户能处理的数据也是不同的。

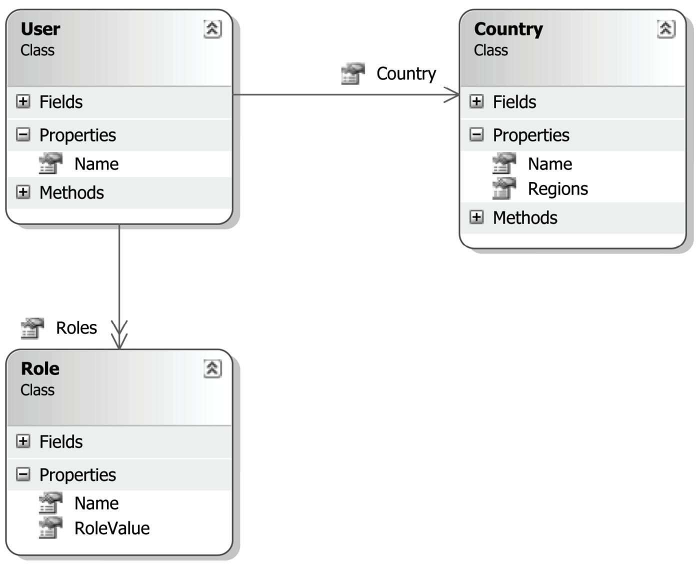


图10-2　用户

显然，除了这些领域对象之外，业务领域还有更多的信息需要被表达。大概你不会感到惊讶，我们将用标注来表达这些信息。

在“Leroy的卡车”里，我们用`DataClassification`标注将领域对象分为4大类。

- 参考数据：国家、地区、仓库、产品、用户。
- 事务数据：时间段、运输、计划的运输。
- 配置数据：角色。
- 审计数据：审计记录。

在实现中，我们用一个枚举来表示这些类别，标注特性唯一的功能就是保存领域对象的分类法。也就是说，即使换成Java实现，代码看起来也会很类似。

```
namespace LeroysLorries.Model.Attributes
{
    public enum DataClassificationValue
    {
        Reference,
        Transactional,
        Configuration,
        Audit
    }

    [AttributeUsage(AttributeTargets.Class)]
    public class DataClassificationAttribute : Attribute
    {
        private DataClassificationValue classification;

        public DataClassificationAttribute(
                                 DataClassificationValue classification)
        {
            this.classification = classification;
        }

        public DataClassificationValue Classification
        {
            get { return classification; }
        }
    }
}
```

最常见的查询操作是找出某个指定类型的分类，为此可以在一个辅助类中创建这样一个方法。

```
public static DataClassificationValue GetDataClassification(Type classToCheck)
{
    return
       GetAttribute<DataClassificationAttribute>(classToCheck).Classification;
}
```

```
private static T GetAttribute<T>(Type classToCheck)
{
    object[] attributes = classToCheck.GetCustomAttributes(typeof(T), true);
    if (attributes.Length == 0)
        throw new ArgumentException( ... );
    return (T)attributes[0];
}
```

上面的公有方法（GetDataClassification）也可以作为特性类本身的一个静态方法来实现，从而使整个API内聚在一处。如果这样实现的话，辅助类中的泛型方法（GetAttribute）就应该变成公有的以便复用。

其他方案

除了使用标注以外，还有其他办法可以给领域数据分类。使用继承是一个显而易见的办法：可以让所有参考数据继承同一个基类（例如ReferenceDataObject），其他领域对象类也如法炮制。但Java和标准的.NET语言不允许多重继承，于是继承就成了张只能打一次的王牌，而我们觉得在别的维度上可能更需要继承。

还有另一重原因——理论性较强但更有说服力——促使我们不用继承来对数据分类：领域驱动设计要求领域专家与技术专家共同拥有领域模型。继承代表了"是一个"（is-a）的关系。如果我们说"地区是一个地点（Location）"，并创建一个Location类来作为Region和Country的父类，那谁也不会有异议；但在真实世界里说"地区是一个参考数据"就毫无意义。简而言之，领域驱动设计的观点认为：领域对象的继承关系应该用于建模真实世界的分类法，而不应该用于其他任何目的。

为了给领域对象加上分类信息，最直观的办法可能就是使用下列接口。

```
interface DataClassification
{
    DataClassificationValue GetDataClassification();
}
```

让所有领域对象都实现这个接口，并把分类信息硬编码在该方法的返回值中。

```
public DataClassificationValue GetDataClassification()
{
    return DataClassificationValue.Transactional;
}
```

走这条路需要更多的代码，而且对于同一类型的领域对象，这个方法所有实例的返回值全都是一个固定的值：它所呈现的是元数据，而不是数据。

可能静态方法更适合这个问题，但Java和C#都不允许接口包含静态方法，而且静态方法也无法实现多态。

另一个办法是使用标记接口（marker interface）——也就是没有任何方法的接口。对于不支持标注的语言，这不失为一条曲线救国之道，但毕竟有悖于语言设计的初衷：接口的原意是用于声明具备多态引用的方法，而这种做法则将其用于保存元数据。

而且如果我们这样使用接口，那就很可能需要创建一个基接口，例如DataClassification，然后用四个子接口分别代表各种分类。但这样一来，我们不仅可以询问一个对象是否拥有ReferenceData接口，还可以询问它是否拥有DataClassification接口。也就是说，领域对象的分类和分类信息的保存被混淆到一起了。

在审计逻辑中的应用

DataClassification第一次被用到是在审计逻辑中，其业务规则是：

- 当且仅当参考数据发生变化时，需要创建审计记录。

这条规则在Auditor类中实现，由这个类负责创建审计记录，它调用了前面介绍过的辅助方法。

```
private bool ShouldAudit(Type type)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(type);
    return classification == DataClassificationValue.Reference;
}
```

用以决定是否应该创建审计记录的信息是以标注的形式保存在领域对象中的，但使用这一信息的逻辑则在Auditor类中。通常应该把数据和行为放在一起，但在领域标注这里，同样的数据可能会被用作多种用途，也就是说如果想把数据和行为放在一起，那么就必须将与某一标注相关的所有行为都放到同一个地方。在下一节里我们会看到，标注的用途可以有很大差异，往往应该将它们彼此分开。

在权限检查中的应用

下列业务逻辑在实现时也用到了DataClassification标注：

- 只有全球管理员能修改参考数据。

PermissionChecker类的一个方法实现了该逻辑，其中也用到了同一个反射辅助方法来判断一个对象是否可以被修改。在辅助方法的帮助下，PermissionChecker中的方法实现极其简单，并且完全聚焦于业务逻辑的实现。

```
public bool CanChangeObject(User user, object anObject)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(anObject.GetType());
    switch(classification)
    {
        case DataClassificationValue.Reference:
            return user.HasRole(RoleValue.GlobalAdmin)
        default:
            return true;
    }
}
```

与前面的代码相比，这段代码的不同之处在于：它并不直接使用标注的值，而是根据数据类别来选择适当的权限判断逻辑。

在数据加载中的应用

如前所述，“Leroy的卡车”是一个智能客户端应用程序，即使与服务器断开连接也可以使用。也就是说，客户端必须在连线时下载部分数据，然后才能脱机使用。为了尽量降低服务器负载和下载流量，要下载的这部分数据应该尽可能小。

基于此，应用程序又在数据分类的基础上实现了以下逻辑：

- 只有计划制订者才下载事务数据。

该逻辑的实现与前面的代码大同小异，但显示出了在领域对象上使用具体方法的标注的优点：我们把领域对象的类型也传入ShouldLoad方法，是因为当这个方法被调用时，领域对象的实例还不存在，该方法正是要判断是否应该创建领域对象实例。

```
private bool ShouldLoad(Type type, User user)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(type);
    if(classification == DataClassificationValue.Transactional)
        return user.HasRole(RoleValue.Planner);
    return true;
}
```

在真实的应用程序中，这部分规则远比“Leroy的卡车”要来得复杂，因此用标注来区分数据类别也就显得更具吸引力。

导航提示

本文将要展示的第二个领域标注是关于领域模型之间的间接关系的。举例来说，尽管“仓库”

与“国家”并没有直接的关系，但我们仍然可以说“某仓库位于某国家”，因为“仓库”与“地区”有关系，而“地区”又与“国家”有关系。显然，类似这样的间接关系不仅仅出现在同一类的对象（例如描述地域信息的对象）之间，例如“运输”同样是与“国家”相关的：“计划的运输”与两个“仓库”相关，于是又与两个“国家”相关。

在“Leroy的卡车”里，我们选择用标注来描述“要到达一个目标对象应该从哪个属性导航”——“国家”就可能是这样一个目标对象。“从某个属性导航”的意思是：从该属性取出另一个领域对象，在后者身上再搜索具有同样标注的属性，如此反复直至到达目标对象。

这个标注的实现甚至比前面的数据分类标注还要简单，因为其中不需要任何值，这也表示它的Java版本会非常类似。

```
namespace LeroysLorries.Model.Attributes
{
    [AttributeUsage(AttributeTargets.Property)]
    public class CountrySpecificationAttribute : Attribute
    {
    }
}
```

将这个特性加诸Warehouse类的一个属性会消减掉任何包含“Warehouse”一词的代码，其形式大致如下。

```
namespace LeroysLorries.Model.Entities
{
    [DataClassification(DataClassificationValue.Reference)]
    public class Warehouse
    {
        private Region region;

        [CountrySpecification]
        public Region Region
        {
            get { return region; }
            set { region = value; }
        }
    }
}
```

为这个标注提供功能逻辑的是泛型的PathFinder类。它能根据给定的对象类型找到想要的目标对象，以及“如何到达这个目标”的路径信息（以字符串数组的形式）。下面的两个例子展示了它的用法。

```
Warehouse warehouse; // get this from somewhere
PathFinder<Country> finder = new PathFinder<Country>();
Country country = finder.GetTargetObject(warehouse);
```

创建了一个目标为Country的PathFinder对象之后，就可以用它来找出某个仓库所在的国

家。值得一提的是，PathFinder用了一个命名约定来判断应该寻找什么标注：它直接在目标类型名（在这里就是“Country”）后面加上“SpecificationAttribute”，然后就到Attributes命名空间去寻找具有该名字的标注特性。稍后我们就会看到，为什么需要创建这样一个泛型的PathFinder类。

```
PathFinder<Country> finder = new PathFinder<Country>();
string[] path = finder.GetPath(typeof(Warehouse));
```

在第二个例子里，PathFinder对象返回了找到Country对象需要经过的路径。路径以字符串数组的形式返回，其中的内容包括“Region”、“Country”等，分别代表了在Warehouse类和Region类中应该访问的属性名字。显然对于同一类型的领域对象，该方法的返回值始终一致，因为它关注的是类型而非实例。

```
public class PathFinder<T>
{
    private static string NAMESPACE = "LeroysLorries.Model.Attributes.";
    private Type attrType;
    public PathFinder()
    {
        string typeName = NAMESPACE + typeof(T).Name + "SpecificationAttribute";
        if((attrType = Type.GetType(typeName)) == null)
            throw new ArgumentException( ... );
    }

    public T GetTargetObject(object anObject)
    {
        Type objectType = anObject.GetType();
        if(objectType == typeof(T))
            return (T)anObject;
        PropertyInfo propInfo = ReflectionHelper.GetPropertyWithAttribute(
                                                       objectType, attrType);
        object nextObject = ReflectionHelper.GetPropertyValue(
                                                    anObject, propInfo.Name);
        return GetTargetObject(nextObject);
    }

    public string[] GetPath(Type type)
    {
        List<string> path = new List<string>();
        if(BuildPath(type, path) == false)
            throw new ArgumentException( ... );
        return path.ToArray();
    }

    private bool BuildPath(Type type, List<string> path)
    {
        if(type == typeof(T))
            return true;
        PropertyInfo prop = ReflectionHelper.GetPropertyWithAttribute(
                                                           type, attrType);
```

```
            if(prop == null)
                return false;
            path.Add(prop.Name);
            return BuildPath(prop.PropertyType, path);
        }
    }
```

其他方案

和前面“数据分类”的例子一样，这里也有其他不使用标注的方案。最显而易见的办法是在每个领域对象上为所有相关对象创建一个属性，将间接关系硬编码进去。

譬如说，可以给Warehouse类加上以下属性。

```
public Country Country
{
    get { return region.Country; }
}
```

这样一来，PathFinder就不再寻找带有“Country”相关标注的属性，而是直接调用返回类型为Country的属性。虽然这仍然很好地实现了面向对象的封装，但为了维持领域模型的一致性，你需要更多的纪律和耐心。

另一种自动化程度较高的办法不需要在领域模型上添加代码，而是直接采用图搜索算法：领域对象类型及其之间的关联可以看作一张有向图，只要从某个领域对象出发进行标准的深度优先或者广度优先搜索，就能找到目标对象。这样的算法不仅适用于类型，也同样适用于实例，因此完全能够在此基础上实现PathFinder的全部功能。

如果把搜索算法发现的路径缓存起来，这种方案就更漂亮了——只要路径没有歧义并且不需要用到额外的领域逻辑，我们就可以这样做。但“Leroy的卡车”偏偏不行：一次计划的运输需要关联到一个来源仓库和一个目标仓库，如果两者位于不同的国家，那么搜索算法就需要额外的信息来判断应该选择哪条路径。使用标注时，我们把CountrySpecification加诸“来源仓库”属性，这就反映出了一项领域知识：运输计划应该按其起点而被划入不同国家。

在权限检查中的应用

再次回到前面曾介绍过的PermissionChecker类：我们可以用CountrySpecification来实现下列业务规则：

- 国家管理员只能修改自己国家的参考数据。

我们对CanChangeObject()方法加以延伸，针对参考数据被修改的情形加上了新的规则，如

下列代码所示。

```
public bool CanChangeObject(User user, object anObject)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(anObject.GetType());
    switch(classification)
    {
        case DataClassificationValue.Reference:
            if(user.HasRole(RoleValue.GlobalAdmin))
                return true;
            if(user.HasRole(RoleValue.CountryAdmin))
                return FindCountry(anObject) == user.Country;
            return false;
        default:
            return true;
    }
}
```

全球管理员仍然可以修改任何参考数据；而对于国家管理员而言，参考数据领域对象所属的国家必须与管理员所属的国家匹配。对PathFinder的调用被抽取到一个辅助方法中，以保持代码整洁。

```
private Country FindCountry(object anObject)
{
    return new PathFinder<Country>().GetTargetObject(anObject);
}
```

这个例子不仅展示了同一个标注的不同用法，而且也展示了如何同时使用不同的标注从而在不损害关注点分离原则的前提下更清晰、更准确地实现业务规则。

在数据加载中的应用

计划制订者被分配到具体国家，并且也只能处理本国的运输事务。与此类似，国家管理员也只能维护本国的数据。考虑到客户端应用的下载流量，显然可以用下列规则来缩减数据下载量：

- 除全球管理员之外，只下载用户所属国家的数据。

借助CountrySpecification和PathFinder，我们就能描述任何一个领域对象如何找到其所属国家。接下来的事就是用O/R映射技术将这一规则转换为查询条件，从而减少读入内存的对象数量。

下列代码展示了这一思路的大致实现。

```
private Query CreateQuery(Type type, User user)
{
    QueryBuilder builder = new QueryBuilder(type);
    if(!user.HasRole(RoleValue.GlobalAdmin))
```

```
    {
        PathFinder<Country> finder = new PathFinder<Country>();
        string[] path = finder.GetPath(type);
        builder.AppendCondition(path, QueryBuilder.EQUALS, user.Country);
    }
    return builder.GetQuery();
}
```

显然，对于给定的类型与用户，只有当前一节所介绍的ShouldLoad()方法返回true时，这个方法才会被调用。

在“Leroy的卡车”中，地理只是重要的维度之一，同等重要的维度还有时间：计划制订者的工作都是以一月为周期的。也就是说，除了历史参考之外，计划制订者只对三个月的数据感兴趣：前月的、当月的、下月的。所以我们不会下载整年的数据，而是按照下列规则来下载数据，对于其他数据则采用延迟加载：

- 只下载前月、当月和下月的事务数据。

前面有另一条规则规定：只有计划制订者才下载事务数据。这条规则是对前者的补充完善。

由于PathFinder在实现时采用了泛型，并且以命名约定来寻找相关标注，因此我们可以创建下列标注特性，并加诸于“运输”和“计划的运输”等领域对象。

```
namespace LeroysLorries.Model.Attributes
{
    [AttributeUsage(AttributeTargets.Property)]
    public class PeriodSpecificationAttribute : Attribute
    {
    }
}
```

在此基础上，我们可以延伸CreateQuery()方法，加入新的规则。

```
private Query CreateQuery(Type type, User user)
{
    QueryBuilder builder = new QueryBuilder(type);
    if(!user.HasRole(RoleValue.GlobalAdmin))
    {
        PathFinder<Country> finder = new PathFinder<Country>();
        string[] path = finder.GetPath(type);
        builder.AppendCondition(path, user.Country);
    }
    if(ReflectionHelper.GetDataClassification(type) ==
    DataClassificationValue.Transactional)
    {
        PathFinder<Period> finder = new PathFinder<Period>();
        string[] path = finder.GetPath(type);
        builder.AppendCondition(path, period);
    }
```

```
        return builder.GetQuery();
    }
```

我相信最后这个例子已经足以展示：借助领域标注和基于这些标注的泛型算法，能够多么整洁地分离各种不同的关注点。对数据层的优化需要用到某些领域知识，这些知识被清晰地分离到了领域模型中实现，而数据访问的逻辑则完全没有漏出数据访问层之外。

而且这个例子把案例分析中谈到的三个标注捏合到了一起，再次展现了用标注来实现横切关注点（cross-cutting concern）的妙处——横切关注点应该由一块独立的代码来实现，标注正好适合这个用途。

10.3　总结

Java里的标注和C#里的特性给这些编程语言增加了一种构造，让开发者能够以一种清晰的可扩展方式来描述元数据。采用模型驱动设计方法时，领域模型的元数据就可以用标注来描述：这就是我们所说的领域标注。这些标注只作用于领域模型，通常与领域模型定义在同一个包/命名空间里，并且常被用作多种用途。

借助领域标注，我们能够更容易地将领域特有的代码与基础设施代码分开，从而分别重用。这种分离的好处是双向的：所有与领域相关的知识都在领域模型中描述，可以在不同的基础设施技术之上重用这个领域模型；基础设施代码本身则可以行业标准化（以商业软件或者开源软件的形式）。最终，我们希望应用开发者能集中关注应用程序的领域逻辑，并创造出价值持久的作品。

第 11 章 重构Ant构建文件

Julian Simpson，构建架构师

在编写和维护构建文件时，所有那些代码完美主义者的理想似乎都消失无踪了……

——Paul Glover

11.1 简介

没有人愿意修改糟糕的软件构建系统。哪怕是一丁点儿的修改也可能让同事不能工作。由于这个“坏名声”，构建文件一旦能够工作，就不会有人主动去做任何改进。这篇文章将向你展示如何重构Ant构建文件，从而减少这种痛苦，并可以对它进一步进行修改。介绍每个重构方法时采用相同的表示方法：重构前后的例子分别用“before”和 “after”标识（用箭头分开，用来显示是如何转换的），后跟一段解释。通过本文的学习，你会掌握一些切实可用的技术，能够使Ant构建文件更简短小巧、更清晰易读、更容易修改。

什么是重构？什么是Ant？

重构是一门艺术，通过一些小的修改，使得代码清晰易读、易于维护。重构不改变任何功能；但可以使代码结构更容易理解。

Ant是许多Java软件项目构建工具的不二选择。Ant创建之时，正值XML风头正劲，那时XML好像能够解决软件开发中的任何问题。所以，使用Ant的项目要依赖一个XML文件，通常称之为`build.xml`。这相当于Java中的Makefile。Ant是个开源项目，非常成功，但是随着项目发展，变得愈加复杂，构建文件通常变得难以维护。

什么时候应该重构？什么时候算搞定了？

在详细介绍每种重构技术之前，让我们看看工作的环境和目的。我们中大多数人的目的是要

交付可以工作的软件。要想交付，你得能够构建它。有时候你需要对构建进行修改，但如果没人愿意修改，你就无法交付。这样就糟了。

所以，这篇文章将尝试如何减少修改构建的成本（或者痛苦）。显然你需要全盘考虑。构建是让你感觉不爽的主要原因吗？是它降低了你交付软件的速度？项目本身的问题是否比构建更大？你现在就得改进构建吗？

如果你还在阅读本文，并且你的构建仍然有问题。那么这个问题有多大？如果它真的是一个意大利面怪兽，这篇文章就是为你准备的，但是要非常、非常小心地进行。首先看看你能删掉什么东西，可能有一些目标从未用过，就从这些目标开始。我曾在某些项目上成功地使用Simian相似性分析器发现了重复，然后使用“提炼宏定义”或者“提炼目标”解决了这些重复。

多数重构方法可以同时存在，不过头几个涉及“提炼”的重构方法在某些情况下是互斥的——它们在效果上完全相同。使用这些重构方法的原因各种各样。你可能想确保代码更容易理解（用“描述”代替“注释”），也可能想防止其他人不经意地使用构建（通过强制使用内部目标）。

你能重构build.xml文件吗

重构时很容易定义构建文件的外部行为。给定一些源文件，通常想生成某些工件——经过编译的代码、测试结果、文档或者可以部署的工件（如WAR文件）。

Ant 构建文件与商业代码一样需要重构。与编程语言相比它们的容错性更差。一些错误不会立刻破坏构建，但是可能会使构建后来莫名其妙地失败。比如，设置属性失败并不会导致构建退出，这与Java、Ruby或者Python中没有声明变量不同。我们遇到的主要困难是，重构时没有任何测试的安全保证或者IDE工具的帮助。

重构通常十分依赖单元测试，确保没有破坏代码的任何功能。也就是说，在对构建文件进行重构时，很少有工具能够帮助我们查看修改带来的任何影响。Ant没有类似Java中的JUnit 或者Ruby中的Test::Unit一样普遍适用的测试框架，因此不能方便地隔离一个单元并使用代理进行测试。即使你有这样的工具，其价值也颇值得怀疑：如果一个构建系统复杂到需要测试驱动的程度，你还会用它吗？

更糟糕的是，Ant有一个静态类型体系，却没有编译时类检查。另外，构建文件采用XML文件描述，由于没有固定的DTD，所以不能进行验证。修改很差劲的构建文件风险很高，哪怕一个变化也会影响生产效率。重构开始时，很难在提交前测试你本地的修改。你可以从很小的修改开始，尽量频繁地测试；随着构建文件内部结构逐渐清晰，你的信心也将逐渐增长，从而可以做一

些幅度更大的重构。

11.2 Ant 重构列表

每个重构包括名称、简单描述和一个实际的重构例子。第一段是原始代码，箭头后面是重构之后的代码。后面接以稍长一些的解释，部分重构以边栏方式对某些问题做了额外的说明。

重　　构	描　　述
提炼宏定义	把Ant中的小块代码提炼成宏定义，并赋予其合适的名称
提炼目标	把较大目标的一部分提炼成独立的目标，并让前一个目标依赖于它
引入声明	让目标声明它们的依赖
以依赖取代调用	使用目标间的依赖取代antcall调用
以属性取代字面量	使用属性取代构建文件中的字面量
引入filtersfile	在filterset元素中使用属性文件而不是嵌套的filter元素
引入属性文件	把build.xml文件中的属性转移到平面文件中
将目标转移到包装器的构建上	把非开发者使用的目标转移到更高一层的文件中，并在该文件中调用开发者构建
以描述取代注释	使用描述特性，而不是XML注释来标注元素
将部署代码放进导入文件	从外部构建文件中导入部署代码，这样你能够在构建时引入正确的文件
将元素移动到Antlib	以Antlib的方式共享项目间经常使用的任务
以fileset取代大型类库定义	使用fileset自动发现所有类库，而不是逐一指定路径
移动运行时属性	构建代码的属性和运行时配置的属性要确保分开
通过ID重用元素	某一类型实例只声明一次（比如，一个fileset），其他地方引用以避免重复
将属性移动到构建正文	把属性放到build.xml的正文部分，以免造成这些属性只属于这个目标的错觉（实际上它们并不是）
以location代替值特性	用location特性表示文件系统的路径，Ant会对路径归一化
将包装器脚本放置到build.xml文件中	把跨平台Ant脚本中的输入验证和类路径操作放置到build.xml中
添加taskname特性	添加taskname特性，从而在运行时显示任务的目的
强制使用内部目标	禁止从命令行调用内部目标
将输出目录移动到同一个父目录	把所有输出与同一个目录下的构建区分开来
以Apply取代Exec	执行时使用类似路径的结构作为输入，而不是一组参数元素
使用CI发布	开发者的构建结束后再为构建打标签，并发布工件，而不要在构建过程中这样做
引入不同的目标命名方式	给目标和属性使用不同的分割符号以提高可读性
用名词重命名目标	用输出结果来给目标命名，而不是使用它所应用的过程

提炼宏定义

摘要：宏定义可以用来梳理构建文件中容易混淆的片段。

下载 refactoring_before.xml

```
<target name="build_and_war_foo.war">
    <javac srcdir="src/foo" destdir="classes/foo" />
    <copy todir="${classes.dir}">
        <filterset>
            <filter token="ENV" value="${environment}" />
        </filterset>
        <fileset dir="config" />
    </copy>
    <war destfile="foo.war">
        <fileset dir="${classes.dir}" />
    </war>
    <move todir="archives" file="foo.war" />
</target>
```

下载 refactoring_after.xml

```
   <macrodef name="build_code">
       <attribute name="component" />
       <sequential>
           <javac srcdir="src/@{component}" destdir="classes/@{component}" />
           <copy todir="${classes.dir}">
               <filterset>
                   <filter token="ENV" value="${environment}" />
               </filterset>
               <fileset dir="config" />
           </copy>
       </sequential>
   </macrodef>

<macrodef name="make_war">
       <attribute name="component" />
       <sequential>
           <war destfile="@{component}.war">
               <fileset dir="${classes.dir}" />
           </war>
           <move todir="archives" file="@{component}.war" />
       </sequential>
 </macrodef>

 <target name="foo.war" >
     <build_code component="foo"/>
     <make_war component="foo"/>
 </target>
```

Ant中的大目标与OO语言中的大方法有着相同的坏味道。它们会变得脆弱、难以测试和调试，

也很难被重用。

由于每一行都可能隐含地依赖其他内容，所以修改起来非常困难。目标太长也会使读者搞不清楚构建文件的作者的真实意图。

宏定义任务是一种容器任务（它嵌套顺序任务或者并行任务，这些任务本身包含你想重用的任务），可以在构建文件中的任何地方调用，并传入属性。属性可以有默认值，当你多次使用某个特定的宏定义时非常方便。

宏定义本身非常易于重用。在前面的例子中，目标做的事情太多，我们可以把其中的一部分抽取成独立的片段，进而可以引入一个新的目标。

下载 refactoring_before.xml

```
<target name="bar.war">
 <war warfile="bar.war" basedir="classes/bar"/>
</target>

<target name="baz.war">
 <war warfile="baz.war" basedir="classes/baz"/>
</target>
```

下载 refactoring_after.xml

```
<macrodef name="war">
 <attribute name="name"/>
    <sequential>
         <war warfile="'@{name}.war" basedir="classes/@{name}"/>
    </sequential>
 </macrodef>
```

构建文件中很容易有重复代码。antcall在Ant旧版本（Ant 1.6之前）中就是为了重用，而宏定义可以很好的代替antcall。比如前面所说的例子就有重复，你可以通过传入不同的属性，调用同一个宏定义来代替它们。

我们写宏定义到什么程度合适呢？如果事情很复杂，可以写比较大的宏任务，但是如果用目标能够更好的表达，就不必引入宏任务。XML语言已经证明很难扩展；你也可能想用Java或者动态语言编写自己的Ant任务。你可以采用Ruby或者Python这样的语言，使用scriptdef任务编写自己的任务，从而享受鱼与熊掌兼得的妙处：你写的代码可以有测试，并且不需要编译。注意你需要花点时间学习Ant的对象模型。

提炼目标

摘要：把执行多个任务的目标分解为2个或者多个目标。

下载 refactoring_before.xml

```
<target name="test" >

   <javac srcdir="${test.src}" destdir="${test.classes}">
      <classpath refid="test.classpath"/>
   </javac>
   <junit failureproperty="test.failure">
      <batchtest todir="${test.results}">
                 <fileset dir="${test.results}"
                    includes="**/*Test.class"/>
          </batchtest>
   </junit>
</target>
```

下载 refactoring_after.xml

```
<target name="compile_tests" depends="compile_code">
   <javac srcdir="${test.src}" destdir="${test.classes}">
      <classpath refid="test.classpath"/>
   </javac>
</target>

<target name="unit_tests" depends="compile_tests">
   <junit failureproperty="test.failure">
      <batchtest todir="${test.results}">
         <fileset dir="${test.results}"
             includes="**/*Test.class"/>
      </batchtest>
   </junit>
 </target>
```

Ant中比较长的目标难以理解，不利于故障查找或者添加新内容。对这种已经存在的长目标，最容易做的短期修改是添加一些新内容，大家也经常这么干。其实如果把这部分功能拆解成多个独立的目标，并使目标具有正确的依赖，可以帮助你保持build.xml文件更干净、更易于维护。

什么时候应该提炼宏定义，什么时候应该提炼目标呢？如果这段代码有所依赖，就使用目标。如果你想从命令行调用，比如，通过调用删除你自己的数据库Schema，也应该使用目标。事实上，几乎所有的时候你都应该使用目标。如果有一些代码块看上去是重复的，或许只是路径或者输入不同，那么可以提炼成宏定义，这样你可以传入不同的特性进行调用。大型项目中的源代码目录很多，测试的类型也很多，对它们的编译以及单元测试，都是很好的例子，可以通过调用宏定义的目标来完成工作。

另外还有一个有用的技巧，当你想使用antcall时，不妨使用宏定义；一个实际的例子是使用antcall调用一个目标，检查构建的状态。antcall会比较慢，因为实际上讲它需要创建一个新的项目对象。

引入声明

摘要：使用Ant内置的声明逻辑代替很难调试的`if`条件。

下载 refactoring_before.xml

```
<target name="deploy">
    <if>
     <equals arg1="${j2ee.server}" arg2="was" />
     <then>
       <antcall target="was_deploy"/>
     </then>
     <else>
       <antcall target="weblogic_deploy"/>
     </else>
    </if>
</target>
```

下载 refactoring_after.xml

```
<property name="j2ee.server" value="was" />
<import file="${j2ee.server}.build.xml" />
<!-- there is now a an appropriate target named
        deploy depending on the version of the app server -->
```

本例中，我们通过判断逻辑调用不同的目标。这虽然感觉比较自然，但是在基于XML的语言中很难表达清楚。XML最初的目的是用来表现数据，逻辑处理并不好用。除此之外，Ant是个声明性的语言。你可以把文件的部分控制权交给Ant，它会按照正确的顺序执行任务——只需给一点点的引导就行。Ant文件中的每个目标使用depends属性声明它的依赖。如果你觉得需要在构建文件中引入分支，这意味着你可能需要重新组织构建文件了。

如果构建文件中有许多`if-else`元素，那么这个技术相当有用； 你可以把这些分支写入到多个清晰简单的文件中。使用面向对象编程的开发者会意识到这就是多态，Ant构建的目标名字相同，但其行为根据环境的不同而不同。

从1.6版起，Ant引入了导入任务，你可以使用这一特性，通过传入不同的属性，导入你期望的文件。

---

ant-contrib

`if`元素来自于ant-contrib项目，它给Ant添加了一些脚本的能力。Ant创始人以及之后的维护者都这么认为：Ant不应该变成一个纯粹的脚本语言。所以小心使用ant-contrib！

以依赖取代调用

摘要：让Ant管理目标间的依赖关系，而不是显式调用别的目标。

下载 refactoring_before.xml

```
<target name="imperative_build">
    <antcall target="compile"/>
    <antcall target="test"/>
</target>
```

下载 refactoring_after.xml

```
<target name="declarative_build" depends="test, publish "/>
<target name="test" depends="compile"/>
```

像Ant这样的构建工具是基于依赖的，首要工作就是防止目标运行多次。antcall以命令方式执行任务，打破了Ant这一微妙的声明特性。通过向调用传入不同参数，它经常用来尝试重用某个任务。使用antcall，很容易多次运行同一个目标，特别当你在构建中混合使用depends和antcall时。

正确的做法是声明deploy目标依赖于编译和测试。测试目标本身也依赖于编译。Ant设计之初就能识别依赖关系树，并能按照正确的顺序执行。它也能尽量按照你声明的顺序执行，但是不要指望过多，因为它会根据依赖关系自动调整执行顺序。

以属性取代字面量

摘要：使用属性代替构建文件中重复的字面量；对外部值可以使用内置的Java和Ant属性。

环境变量

一个常见的味道是使用导入的环境变量查找用户名或者操作系统。虽然这样也能起作用，但是产生了对构建系统的外部依赖，并且导致系统比较脆弱。默认情况下Java的系统属性被包含在“Ant构建”的命名空间中。所以不要尝试从导入的环境变量中查找所需条目，而是尽可能使用内置的属性：如user.name，os.name等。

下载 refactoring_before.xml

```
<target name="deploy_to_tomcat">
    <copy file="dist.dir/webapp.war" todir="tomcat.webapps.dir"/>
</target>
```

下载 refactoring_after.xml

```
<property name="dist.dir" location="${build.dir}/dist"/>
<property name="tomcat.webapps.dir" location="/opt/tomcat5/webapps"/>
<target name="deploy_to_tomcat">
    <copy file="${dist.dir}/webapp.war" todir="${tomcat.webapps.dir}"/>
</target>
```

使用属性表示构建文件中的静态和动态字符串非常有必要。比如类文件的编译目录很少要修改，但是一旦要修改，你只想进行尽可能少的修改。一般说来，如果输入同样的字符串达到3次，你就应该使用属性。要想更多地了解如何在属性中表示文件系统的路径，请查看“以Location取代值特性”一节。应该谨记，Ant 中的属性是不变的。所以，第一次赋给属性的值维持不变。这意味着，你可以重写构建文件外部定义的属性，或者在属性文件中引入一些默认值，以便之后再使用。

引入filtersfile

摘要：通过属性文件建立元素与值之间的映射关系，在模板中直接引用属性文件，而不是分别定义这些属性。

下载 refactoring_before.xml

```
<target name="filter">
        <copy todir="${build}" file="${src}/config/config.xml">
            <filterset>
                <filter token="APP_SERVER_PORT" value="${appserver.port}"/>
                <filter token="APP_SERVER_HOST" value="${appserver.host}"/>
                <filter token="APP_SERVER_USERID" value="${appserver.userid}"/>
                </filterset>
    </copy>
</target>
```

下载 refactoring_after.xml

```
<target name="filtersfile">
    <copy todir="${build}" file="${src}/config/config.xml">
        <filterset filtersfile="appserver.properties"/>
     </copy>
</target>
```

下载 appser ver.properties

```
appserver.port=8080
appserver.host=oberon
appserver.userid=beamish
# END filtersfile
```

构建文件很快就会变得难以阅读。有时最好把属性放到纯文本文件中，这样项目团队中的任何一个人都可以看到，并明白是什么意思。上面以filtersets为例演示了如何使用。许多构建系统用值代替模板中的标记，尤其当你需要为不同的环境维护多个文件时。如果你还没有通过ID重用filterset元素，你会发现构建文件中充斥着大块的标记。引入filtersfile这种方式有两个好处：首先你不需要给值引入一个标记（你可以直接使用属性名称），其次你可以把属性文件公开给大家，每个人都可以编辑XML文件而不会使其失效。当然你也可以给副本元素的子元素使用不止一个filtersfile值。使用时遵循“先到先服务”的原则，所以可以设置默认值。筛选器文件是一个纯属性文件，可以在构建中的其他地方使用。

引入属性文件

摘要：把主构建文件中很少变化的属性移入到文件中。

应该有多少属性才合适呢？

属性太少，会形成令人讨厌的属性串联，与不要重复自己（DRY）的原则相冲突；太多了，一些属性可能重复，或者很难记住。如果能够把构建分成不同的文件，你就能根据文件来划分属性。

下载 refactoring_before.xml

```
<property name="appserver.port" value="8080" />
<property name="appserver.host" value="oberon" />
<property name="appserver.userid" value="beamish" />
```

下载 refactoring_after.xml

```
<property file="appserver.properties" />
```

下载 appserver.properties

```
appserver.port=8080
appserver.host=oberon
appserver.userid=beamish
# END filtersfile
```

Ant 构建文件不支持常量——这毫无必要，因为所有属性在任何情况下都是不变的。你可能遇到这样的区别：一些属性是固定不变的；另外一些属性是动态创建的，比如在 Ant调用时产生或者作为一个任务的输出结果。那些静态属性与常量非常像，你可以把它们从Ant文件中移入到属性文件中。虽然把属性放入文件中会有一些代价，比如无法在构建文件中看到属性的值，但这

确实会使你的build.xml文件变得更清晰易读。

将目标转移到包装器的构建上

摘要：把持续集成的目标从开发者构建文件中拿出来；间接调用开发构建。

下载 refactoring_before.xml

```
<target name="build">
       <!-- developer build-->
    </target>
<target name="functest">
       <!-- functional tests-->
</target>

<target name="cruise" depends="update,build,tag"/>
<target name="functional_cruise" depends="update,build,functest,tag"/>
```



下载 refactoring_after.xml

```
<target name="build">
   <!-- developer build-->
</target>
<target name="functest">
   <!-- functional tests-->
</target>
```

下载 ccbuild.xml

```
<project name="cruise" default="tag">

    <target name="tag" depends="build">
        <!-- code to tag the files you have checked out -->
    </target>

    <target name="build" depends="update">
        <ant buildfile="build.xml"/>
    </target>

    <target name="update" >
        <!-- code to update from your scm system-->
    </target>

</project>
<!-- END ccbuild  -->

<project  default="update" basedir="."
        xmlns:my="antlib:com.thoughtworks.monkeybook">
    <target name="update" depends="build">
        <my:svn_up/>
    </target>
```

```
</project>
<!-- END antlibccbuild  -->
```

持续集成可以极大减少软件项目集成的痛苦。每当有开发者向代码库中提交源码时，持续集成服务器就会检出最新的代码、编译代码并运行测试。如果构建状态发生了变化，可以有多种方式通知团队（对坐在一块的团队来说通过声音发出警告就不错），应该让整个团队积极保持构建为绿色或者通过的状态。如果一段新代码不能与现有的代码集成，团队应该很快就能知道。

持续集成的操作（比如，SCM打标签、更新等）有时与构建变得紧密耦合。理想情况下，持续集成系统与开发者应该运行相同的构建，一些额外的目标包装在Ant文件中，通过`ant`任务调用开发构建。如果你在同一个代码库上运行多个持续集成构建，你可以维护多个`ccbuilds`，能够保持所有这些文件中的持续集成都是安全的。

以描述取代注释

摘要：在标签上使用描述，而不是行内注释。

下载 refactoring_before.xml

```
<target name="distribute">
        <!-- copy the compiled classes -->
    <copy todir="${dist}">
        <fileset dir="${classes.dir}"/>
    </copy>
</target>
```



下载 refactoring_after.xml

```
<target name="dist">
        <copy todir="${dist}" description="copy compiled classes">
        <fileset dir="${classes.dir}"/>
    </copy>
</target>
```

许多Ant构建文件写满了注释。注释可能是个好东西，但是它们也会使构建晦涩难懂。几乎所有的任务都接受`description`特性；所以，你可以直接给任务添加注解，而不是就近引入注释。你也可以使用`taskname`特性告诉使用者在运行时会发生什么。我喜欢保持任务名称简短，所以把长长的解释放到描述中。

将部署代码放进导入文件

摘要：把部署代码的目标与开发的目标放入不同的文件。

尽量仿真

这一点总是值得强调：尽量部署到与生产环境类似的开发/测试环境。在选择轻量级容器时应该充分考虑这方面的因素。

下载 refactoring_before.xml

```
<target name="deploy_to_weblogic" >
    <!-- insert WL task or similar -->
    <sshexec host="${deploy.host}" username="dev" command="restart_container"/>
</target>
```

⇩

下载 refactoring_after.xml

```
<import file="deploy.xml" />
<target name="test_in_container" depends="deploy_to_weblogic"/>
```

如果所有的项目只在一个主机上使用一种应用程序服务器，那么只用一个构建文件比较简单。但通常项目使用不止一个应用程序服务器，比如在开发机器上使用轻量级的服务器，在数据中心使用企业级服务器。

如果把服务器相关的代码提炼到单独的文件中，就能实现很好的分离。你可以根据需要导入相关的文件，所有与部署相关的细节将被隐藏起来。除此之外，当Ant解析构建文件时能够大大简化最后的项目对象，所以当你想部署到企业级服务器时，丢掉本地构建的某些依赖并不是个问题。

将元素移动到antlib

摘要：把多个项目中重复的Ant构建元素拿出来，通过antlib分发。

下载 ccbuild.xml

```
<project name="cruise" default="tag">

    <target name="tag" depends="build">
        <!-- code to tag the files you have checked out -->
    </target>

    <target name="build" depends="update">
        <ant buildfile="build.xml"/>
    </target>

    <target name="update" >
        <!-- code to update from your scm system-->
    </target>
```

```
</project>
<!-- END ccbuild  -->

<project  default="update" basedir="."
        xmlns:my="antlib:com.thoughtworks.monkeybook">
    <target name="update" depends="build">
        <my:svn_up/>
    </target>
</project>
<!-- END antlibccbuild  -->
```

下载 ccbuild.xml

```
<project  default="update" basedir="."
        xmlns:my="antlib:com.thoughtworks.monkeybook">
    <target name="update" depends="build">
        <my:svn_up/>
    </target>
</project>
<!-- END antlibccbuild  -->
```

下载 antlib.xml

```
<antlib>
        <macrodef name="svn_up">
                <attribute name="svn.exe" default="/usr/bin/svn" />
                <sequential>
                        <echo message="${basedir}" />
                        <exec failonerror="true" executable="@{svn.exe}">
                                <arg value="update" />
                        </exec>
                </sequential>
        </macrodef>
</antlib>
<!-- END antlib-->
```

基于Ant的项目经常一遍又一遍地重复着相同的XML代码。这也是导致Maven项目出现的一个原因："不同的构建有很多相似之处，每个社区创建自己的构建系统，项目之间没有重用构建逻辑" [Casey]。`antlib`是一个根节点为`antlib`的XML文件。当这个文件在你的类路径中时（可能在你的`$ANT_HOME/lib`目录下，打包成一个JAR文件），并在构建文件中指定XML命名空间，你可以直接访问定义的元素。这是一个现实世界中的例子。

比如，在大型项目中，你可能会有几十个用CruiseControl构建的小项目。每个项目都需要这样做：

- 从代码库中更新代码；
- 调用开发构建；
- 如果构建通过给代码库打标签。

每个类库在哪里使用？

如果你已经在给运行时和构建时的类库创建不同的子目录，这就是很好的开始。知道构建或者部署需要什么东西非常有用。

每个构建都有一个简短的构建文件（可能叫做cc-build.xml或者类似的名字），在调用开发构建之前会执行这些操作。antlib加入默认的类路径后，可以把你定义的类型、任务以及宏定义公开给他人。所以在该例中，你可以声明一个SVN任务或者宏定义，并把它放到$ANT_HOME/lib目录，这样任何其他人或者事物都可以使用这些通用的类型。在团队的其他人使用之前，你需要首先把它打包。

```
mkdir -p com/thoughtworks/monkeybook/
cp ~/workspace/monkeybook/content/antlib.xml com/thoughtworks/monkeybook/.
jar cvf antlib.jar com
cp /tmp/antlib.jar apache-ant-1.6.5/lib/.
```

一旦把JAR文件放到了项目的类路径，你可以像更前面的例子一样使用宏定义。

以fileset取代大型类库定义

摘要：定义路径时使用嵌套的fileset，而不是费劲地手工指定每一个路径元素。

下载 refactoring_before.xml

```
<path id="build.path">
    <pathelement location="${lib}/build/junit.jar"/>
    <pathelement location="${lib}/build/crimson.jar"/>
    <pathelement location="${lib}/build/emma.jar"/>
</path>
```

↓

下载 refactoring_after.xml

```
<path id="build.path">
 <fileset dir="${lib}/build" />
</path>
```

大多数项目把类库检入代码库中。当类库发生变化时更新所有的引用是件麻烦事儿。这个例子中的路径没有版本号，但是一旦你想更新某个类库版本，你不得不修改构建。让Ant自动发现类库帮你省了很多事儿，但是要小心：你仍然需要明白代码使用了什么类库，并用正确的方式组织这些类库。

移动运行时属性

摘要：把运行时的属性与构建的属性分开，以方便你重新配置应用程序。

下载 refactoring_before.xml

```
<target name="war">
    <copy file="${src}/runtime.properties"
        tofile="${build}/war/lib/myapp.properties"/>
    <war destfile="${dist}/myapp.war" basedir="${build}/war"/>
</target>
```

↓

下载 refactoring_after.xml

```
<property name="runtime.smtp.server" value="foo.thoughtworks.com"/>
<property name="web.service.endpoint" value="bar.thoughtworks.com/axis"/>
<target name="war">
    <echoproperties destfile="${build}/war/lib/myapp.properties"/>
    <war destfile="${dist}/myapp.war" basedir="${build}/war"/>
</target>
```

当部署到不同的环境中时，你想重新打包，甚至重新编译应用程序吗？你应该能够编译候选发布版本，把它们存放到某个地方，以后可以用来部署。这保证了部署代码与经过测试的代码总是相同的。我们曾遇到的另外一个问题是，由于程序运行时使用的属性发生了微小的变化，需要重新构建整个应用程序（“Web服务端点的URL变化了？你得重新编译。”）。

将配置解放出来

一旦构建变得慢起来，以及（或者）离发布越来越近，你可以考虑把配置从代码中分离出来。总会有这样的时候：你想修改某个配置属性，但不想重新部署。比如，你可以把运行时的属性文件部署到文件系统上。这么做很好，因为你需要时就可以编辑它们，然而在产品环境中，你需要确保它们是安全的。LDAP可能是另外一种选择（虽然需要更多的工作），你也可以引入另外一个服务，让应用程序依赖它。

构建中有两种不同的属性：构建时属性（编译到什么地方，发布到哪里）以及运行时属性（使用什么数据库证书，外部服务的信息）。很容易一时高兴就把这些运行时属性和构建时属性混在一起。一般情况下这不会引起问题，直到随着环境的变化你开始在代码分支之间反复合并属性，或者仅仅是因为一个属性变化了，你为了得到一个可以发布的版本，需要花费20分钟时间等待自动化的功能测试通过。

如果在软件项目的整个生命周期中都不整理属性，它们最终可能会失控；为自己着想，把它们分开吧——创建可部署应用程序工件（artifact）需要的属性以及运行应用程序所需要的属性。考虑把运行时的属性移到自己的存储库中，这样它们就不会依赖于项目、团队或者代码的特定构建。

通过ID重用元素

摘要：定义某个元素，在其他地方引用，就可以代替重复的元素（如path和set）。

下载 refactoring_before.xml

```
<target name="copy_and_filter">
   <copy todir="${build}/content">
      <fileset dir="${html}" />
      <filterset>
         <filter token="AUTHOR" value="${author.name}" />
         <filter token="DATE" value="${timestamp}" />
         <filter token="COPYRIGHT" value="${copyright.txt}" />
      </filterset>
   </copy>
   <copy todir="${build}/abstracts">
      <fileset dir="${abstracts}" />
      <filterset>
         <filter token="AUTHOR" value="${author.name}" />
         <filter token="DATE" value="${timestamp}" />
         <filter token="COPYRIGHT" value="${copyright.txt}" />
      </filterset>
   </copy>
</target>
```

下载 refactoring_after.xml

```
<filterset id="publishing_filters">
   <filter token="AUTHOR" value="${author.name}" />
   <filter token="DATE" value="${timestamp}" />
   <filter token="COPYRIGHT" value="${copyright.txt}" />
</filterset>

<target name="copy_and_filter">
   <copy todir="${build}/content">
      <fileset dir="${html.content}" />
      <filterset refid="publishing_filters"/>
   </copy>
   <copy todir="${build}/abstracts">
      <fileset dir="${abstracts}" />
      <filterset refid="publishing_filters"/>
   </copy>
</target>
```

许多顶级的元素，如`path`、`filterset`以及`fileset`允许作者通过引用调用。你不必重复定义一个路径，只需声明它一次，赋给它一个ID，然后在`build.xml`文件的其他地方引用它。当你面对一个庞大的`build.xml`文件时这尤其有用。文件行数太多会使人难以理解它的意图；把许多`path`或者`filterset`的声明折叠成一行是件鼓舞人心的事，它能让你得到成就感。

你也可能因此发现一个bug，例如可能有人忘记更新某个很大的build.xml文件中的所有filterset实例。

将属性移动到构建正文

摘要：把目标中声明的属性移到构建文件的主体中。

下载 refactoring_before.xml

```
<target name="distribute">
    <property name="dist_file" value="widget-1.0.tar"/>
    <tar destfile="${dist_file}">
            <tarfileset dir="${build}/dist"/>
    </tar>
    <gzip src="${dist_file}"/>
    <scp file="${dist_file}.gz"
        todir="${appserver.userid}@${appserver.host}:/tmp"/>
 </target>
```

↓

下载 refactoring_after.xml

```
<property name="dist_file" value="widget-1.0.tar"/>
 <target name="distribute">
     <tar destfile="${dist_file}">
             <tarfileset dir="${build}/dist"/>
     </tar>
     <gzip src="${dist_file}"/>
     <scp file="${dist_file}.gz"
         todir="${appserver.userid}@${appserver.host}:/tmp"/>
  </target>
```

多数人都知道如何在代码块中把变量局部化。局部变量这种语法域在Ant中并不存在。如果你声明了一个属性，那么你项目命名空间中的任何任务或者目标都能立刻使用它。很多人习惯在使用属性的地方声明属性，但是不要认为这些属性只局限于这个目标。

真正让你同事感到困扰的是他们是否使用了相同的属性名称；属性的值随着目标的执行顺序会发生变化。使用ant-contrib类库，有可能打破属性值保持不变这一基本特性。这导致你的项目把Ant作为脚本语言而不是构建工具。

以location代替值特性

下载 refactoring_before.xml

```
<property name="libdir" value="lib" />
```

```
<property name="libdir.runtime" value="${libdir}/runtime" />
```

↓

下载 refactoring_after.xml

```
<property name="libdir" location="lib" />
<property name="libdir.runtime" location="${libdir}/runtime" />
```

常有这样的情况：必须在某个特定目录运行，或者设置了特定的环境变量集，你的Ant构建才能工作。你甚至可能有一个wiki页面来告诉新同事如何搭建并构建工作环境。

然而，理想情况下，如果build.xml自己无法找到所需的东西，就应该告诉用户。在属性元素上使用location特性就朝“健壮构建”的目标迈出了第一步。大家可能误用你的脚本和工具；尽量兼容常见的误用情形，如果用户使用不当就立即退出，同时给出友好的错误消息。当你在一个错误的目录下调用时Ant通常做正确的事情；默认情况下，它将${basedir}属性设置为build.xml文件的安装目录。Ant用location特性构造一个属性集，从而与${basedir}目录有所关联，而且通过设置路径为build.xml文件所在目录的完整路径（而不是相对路径），Ant会做正确的事情。许多构建在属性中使用了value特性，会导致构建比较脆弱。

你也可以在其他类型的元素中使用location属性，最值得一提的是传递给类似execute这样的任务的arg元素。它们的作用都一样：给任务提供准确的路径。

将包装器脚本放置到build.xml文件中

摘要：把带有路径和选项的脚本向下移回构建文件中。

下载 go.bat

```
@echo off

set CLASSPATH=%CLASSPATH%;lib\crimson.jar
set CLASSPATH=%CLASSPATH%;lib\jaxp.jar
set CLASSPATH=%CLASSPATH%;lib\ojdbc14.jar
cd build
ant -f build.xml
rem END push_down_wrappers
```

↓

下载 refactoring_after.xml

```
<classpath id="classpath" description="The default classpath.">
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="jaxp.jar"/>
    <include name="crimson.jar"/>
```

```
    <include name="ojdbc14.jar"/>
  </fileset>
</classpath>
```

许多项目最后还用脚本封装了Ant构建，如DOS批处理、Unix shell或者Perl脚本。这些脚本通常包含项目特有的信息，可能还有一些选项；它们可以给复杂项目提供友好的前端界面，让每个人都能构建代码。这也有助于其他脚本进一步封装它们。然而如何调试这些脚本则让人生畏，所以要尽可能避免使用它们，必要时可以使用只有一行的脚本。而且它们也会带来意外的后果，即从不把返回值传递给调用程序。如果你的自动化部署流程调用了Ant脚本，然后做了些其他事情，出错时将难以发现问题所在。

为了使构建更加健壮，你可以使用`fail`元素提前检查，确保所有的属性设置正确。你也可以结合`fail`使用`available`任务，根据你的Ant程序能否获得文件、类路径或者JVM资源等而设置属性。你也可以在命令行使用`-D`选项，创建你可能需要的属性。

如果你受够了在命令行中输入，或者遇到了来自团队其他人的抱怨，那当然也可以创建一个封装的脚本，比如`go.bat`，但务必确保它只有一到两行——从操作系统的shell转移到Ant中，这就足够了。Windows操作系统中需要做一些额外工作：你需要取消CLASSPATH环境变量，以免它污染构建的类路径。许多Windows安装包会在CLASSPATH环境变量前面或者后面添加记录，这在某些情况下会带来不可预知的结果。

然而，Ant执行某些任务时必须使用特定的类库。Ant手册上列出了对外部类库有依赖的任务。

调用Ant

在Unit系统上构建失败时，Ant肯定会给一个非零的退出编码。Windows上你不会这么幸运了，因为某些版本的ant.bat文件臭名昭著，根本不会返回任何Windows错误级别的值。也有一些用于的Ant动态语言封装器；不过你最好测试一下，确保构建失败后会让部署脚本停止运行。

为了满足依赖，你可以使用封装脚本，把需要的类库添加到类路径中，或者你可以把类库放到Ant的`lib`目录。当Ant被调用时，它通过调用`launcher`类从发现的类库中生成类路径。这个方法非常方便，可以满足对类库的依赖。把Ant的这个副本检入到代码库中也是个不错的主意，这样每个人都可以把它检出并运行构建，而不必在他们的电脑上再次搭建项目环境。

添加taskname特性

摘要：让Ant任务显示有意义的任务输出信息，这样你可以理解代码的意图。

下载 refactoring_before.xml

```
<target name="copy_config">
    <copy tofile="${output}/style.xsl" file="${src}/xsl/style.xsl" />
    <copy todir="${output}">
            <fileset dir="${xml.docs}"/>
    </copy>
    <copy todir="${output}/images">
        <fileset dir="${common}/images"/>
    </copy>
</target>
```

↓

下载 refactoring_after.xml

```
<target name="copy_config">
    <copy tofile="${output}/style.xsl"
        file="${src}/xsl/style.xsl" taskname="copy xsl stylesheet"/>
    <copy todir="${output}" taskname="copy xml docs to output">
        <fileset dir="${xml.docs}"/>
        </copy>
        <copy todir="${output}/images" taskname="copy images to output">
        <fileset dir="${common}/images"/>
        </copy>
</target>
```

有时候，你写的构建在同一行多次执行同样的任务；copy就是一个很好的例子。这种情况下使用者很难理解构建在做什么。如果你加上任务名称会很有帮助，让人明白你想得什么东西。

强制使用内部目标

摘要：让内部目标的名称以连字号（-）开头，这样便不能通过命令行调用它们。

下载 refactoring_before.xml

```
<target name="init">
    <mkdir dir="build"/>
</target>
```

↓

下载 refactoring_after.xml

```
<target name="-init">
    <mkdir dir="build"/>
</target>
```

调用错误的目标可能给你的构建带来意想不到的后果，尤其是如果你从来没想到通过命令行运行这些目标。因为与Java代码可以声明私有方法不同，你不能在Ant中将任务声明为私有。于是有人想出了这个简单但是聪明的方法：让Ant误以为这个目标是个参数。shell把它当作位置参

数传递给Ant封装脚本，而 Ant把以连字号开头的参数当作一个选项。所以，Ant别无选择，只能把类似-create_database这样的任务当作一个选项，而这个选项并不存在。当然，如果你的内部目标叫做-propertyfile或者-logger，你就把事情搞乱了。

将输出目录移动到同一个父目录

摘要：把多个地方创建的组件集中起来，放到同一个目录下。

```
project/
|-- build
|-- dist
|-- docs
|-- src
`-- testresults
```

```
project/
|-- build
|   |-- dist
|   |-- docs
|   `-- testresults
`-- src
```

如果代码会动态更新多个目录，事情就变得令人费解。你想从头开始，就必须同时清除多个目录。如果从某个目录挑了一些东西并放到其他的目录，事情就愈加复杂了。应该选择一个目录，在这里生成所有的东西，确保源码控制系统忽略这些产生的文件，并在这里集成所有的构建工件。如果你的属性指向一个规范的位置，移动目录就只需要修改一行代码。

以Apply取代Exec

下载 refactoring_before.xml

```
<exec executable="md5sum" output="md5sums.txt">
        <arg value="${dist.dir}/foo.dll"/>
        <arg value="${dist.dir}/bar.dll"/>
</exec>
```

下载 refactoring_after.xml

```
<apply executable="md5sum" output="md5sums.txt">
        <fileset dir="${dist.dir}" includes="*.dll"/>
</apply>
```

Exec能够使用的参数非常有限，只能用于简单的命令。Apply允许你以Ant的方式执行，这意味这你可以传给它filesets，包括refids等参数，这样就简单多了。

使用CI发布

摘要：不要因为失败的标签而破坏了整个构建；这妨碍了对开发者的及时反馈。

下载 refactoring_before.xml

```
<target name="cruisecontrol" depends="developer_build, functional_tests, tag">
```

↓

下载 refactoring_after.xml

```
<target name="cruisecontrol" depends="developer_build, functional_tests">

<target name="tag"
    description="this will fail unless run from a cruise publisher">
    <fail unless="logdir"
        message="${logdir} property missing -
        are you running this from a cruisecontrol publisher"/>
</target>
```

引入不同的目标命名方式

摘要：让目标和属性有不同的命名风格。

下载 refactoring_before.xml

```
<property name="build.dir" location="${basedir}/build"/>
<property name="lib.dir" location="${basedir}/build"/>
<target name="test.unit" >
    <junit haltonerror="false" haltonfailure="false">
        <!-- details excluded -->
     </junit>
</target>
```

↓

下载 refactoring_after.xml

```
<property name="build.dir" location="${basedir}/build"/>
<property name="lib.dir" location="${basedir}/build"/>
    <target name="unit-test" >
            <junit haltonerror="false" haltonfailure="false">
                <!-- details excluded -->
             </junit>
    </target>
```

“Ant本身的一些特性使人们对待它的方式与对待它所构建的代码的方式非常不同。与一致性、测试及可维护性相关的所有规则，甚至一些常识看上去都不再管用”[Newman]。XML没必要易读。`build.xml`文件则需要尽可能地保持清晰易读。从实践上来说，这意味着`build.xml`文件

应该坚持一种风格。如果目标和属性都采用点来分割单词，很容易让读者困惑：他看不清被引用的究竟是一个属性还是一个值。考虑使用点来分隔属性中的单词，因为Java系统属性是用点分隔的，在Ant项目中可以访问Java系统属性的命名空间。下划线或者破折号则非常适用于目标名称。可以参考“Ant元素的样式”[①]一文以了解更多内容。现在不再需要一些建议了；比如，现在的IDE能够帮助你在Ant文件中导航，不必再加上大段的注释来标识目标。除此之外，类似Grand（您可以查看“资源”一节以了解更多信息）的工具也能帮助你形成构建依赖关系树的图像。

用名词重命名目标

摘要：使用生成的结果给目标命名，而不是用生成结果的过程命名。

下载 refactoring_before.xml

```xml
<target name="foo-build-webapp">
    <war destfile="foo.war">
      <fileset dir="${build.dir}/frontend/dist"/>
    </war>
</target>
```

下载 refactoring_after.xml

```xml
<target name="foo.war">
    <war destfile="foo.war">
      <fileset dir="${build.dir}/frontend/dist"/>
    </war>
</target>
```

“我建议给目标选择的名字要能描述它们生成了什么，比如类、测试/报告。”[Williams]

在Ant `build.xml`文件中有一种习惯，即应该有一个“compile”目标、一个“test”目标等。这其中有特定的逻辑。有时候你只是想运行一个表示状态的目标。这可能是部署准备完毕、发布准备完毕或者类似的事情。你的构建在哪儿产生工件，你就应该关注那个工件。也有可能使用`uptodate`任务来忽略那些不需要运行的目标，节省大家的时间。这也大大提高了构建的清晰度。

如果你曾用过make家族的任何一个成员，你可能之前见过这个。广为接受的make样式就是用生成的组件来命名目标。在得到易用性的同时，我们也不幸失去了平台的独立性。

11.3　总结

这篇文章介绍了如何重构Ant构建文件。一些重构方法直接来自最早的重构惯例[FBB+99]；

① http://wiki.apache.org/ant/TheElementsOfAntStyle

其他的则是源自实践。当你想修改已经存在的构建文件时，开始时可能非常困难，但是付出必有回报。不要忘了，你构建软件是为了把它部署到产品环境。如果一直用这样的标准来考量和改进你的构建系统，发布上线的那个周末将更有可能变得轻松而愉快。

11.4 参考文献

[Hunt, Thomas] *The Pragmatic Programmer*

[Newman] http://www.magpiebrain.com/blog/2004/12/15/ant-and-the-use-of-the-full-stop/

[Casey] *Better Builds with Maven*

[Fowler, Foemmel] http://martinfowler.com/articles/continuousIntegration.html

[Loughran, Hatcher] *Java Development with Ant*

11.5 资源

Grand : www.ggtools.net/grand

The Elements of Ant Style: wiki.apache.org/ant/TheElementsOfAntStyle

11

第 12 章

一键发布

Dave Farley，技术主管

12.1 持续构建

敏捷软件开发的核心实践之一就是持续集成（continuous integration，简称CI）。CI是指开发人员一旦将代码提交到版本控制系统之后，就进行构建，并运行一系列自动化测试套件的过程。

这一实践已被采纳多年，用于提供软件的高安全性：任何时候，正在开发的软件都要进行构建并通过所有的单元测试。这显著提高了团队对其开发的软件达到其质量要求的信心。对许多项目（甚至大多数项目）来说，这是最终交付的软件在质量和可靠性上向前跨进的一大步。

但对复杂项目来说，问题远不止是能否成功编译和通过单元测试这么简单。

较高的单元测试覆盖率当然好，它比项目需求更接近问题的解决方案，然而从某些方面来说，单元测试可以证明所写的产品代码的确是开发团队心中所想的解决方案，而并不会证明产品代码满足业务需求。

一旦软件被开发出来，它就要被部署。而对于现在大多数软件来说，部署过程不再是复制一个可执行文件就万事大吉了：除了软件本身之外，还需要对很多技术事项（如Web服务器、数据库、应用程序服务器和消息中间件等）进行部署及配置。

这样的软件通常都会有一个相当复杂的发布过程，会被部署到各种各样的环境中。比如，它首先可能会被部署在开发环境中运行，然后会被部署到QA环境，接着可能被部署到性能测试环境和试运行环境，最后才是生产环境。

绝大多数（“绝大多数”，如果不是“所有”的话）项目的这一过程都会有相当程度的人为干

预。人们会手工管理配置文件，对那些需要部署的软件进行剪裁，来适应当时的部署环境。此时，人们常会漏掉一些操作步骤或忘记某些文件的所在位置，比如，你常常会听到这样的话："我花了两个小时的时间才发现这个测试环境中的模板文件与生产环境中的位置不同"。

上面提到的持续集成是非常有用的实践，但叫它"持续集成"有些不恰当，可能"持续构建"更为恰当一些。那么，它在整个软件发布过程中处于什么位置？而整个系统的持续集成又应该是什么样子的呢？

12.2　超越持续构建

在过去几年实践中，我所在的团队建立了一个端到端的持续集成发布系统，只要轻轻点击一下鼠标，就可以将大型复杂系统部署到任何一个我们想要部署的环境。这种方法大大减轻了发布时的压力，而且遇到的问题也减少了。在建立这种端到端的持续集成环境的过程中，我们总结出了构建过程的一般步骤，这让我们在项目一开始就可以很快建立这个持续集成系统。

这个流程的核心思想是：每个版本都要通过一系列的检验阶段来证明其质量；每成功通过一个阶段，就证明该版本的软件质量可靠性就增加了一些；　通过最后一个检验阶段的版本就是可以被认为是满足发布条件的候选发布版本。对于敏捷软件项目，每次代码提交都会让其产生一个版本，而每个版本都有可能成为满足发布条件的候选软件。

在一个候选发布版本需要走过的整个流程中，有些环节对于大多数项目来说都是必要的，有些环节则可以根据项目的具体情况加以调整。

我们通常将这一流程及其中的所有环节并称为*构建管道*、*管道式构建*或者*持续集成管道*，此外它也被称为"*分阶段构建*"。

12.3　全生命周期的持续集成

在图12-1中是一个典型的全生命周期，它是一个持续集成管道，也是本方法的关键所在。经过各种不同类型的项目实践，最终证明，该方法具有普遍代表性。当然和所有敏捷实践一样，通常在具体项目中还需要剪裁和调整。

该管道式持续集成过程的起点是代码被提交到代码仓库的那一刻。持续集成工具（我们通常会使用CruiseControl）会发现这次提交，并触发以后的各个阶段：编译代码、运行自动化测试，

如果测试通过，就会将其打包生成二进制文件[①]，并将其保存到统一管理的二进制文件库中。

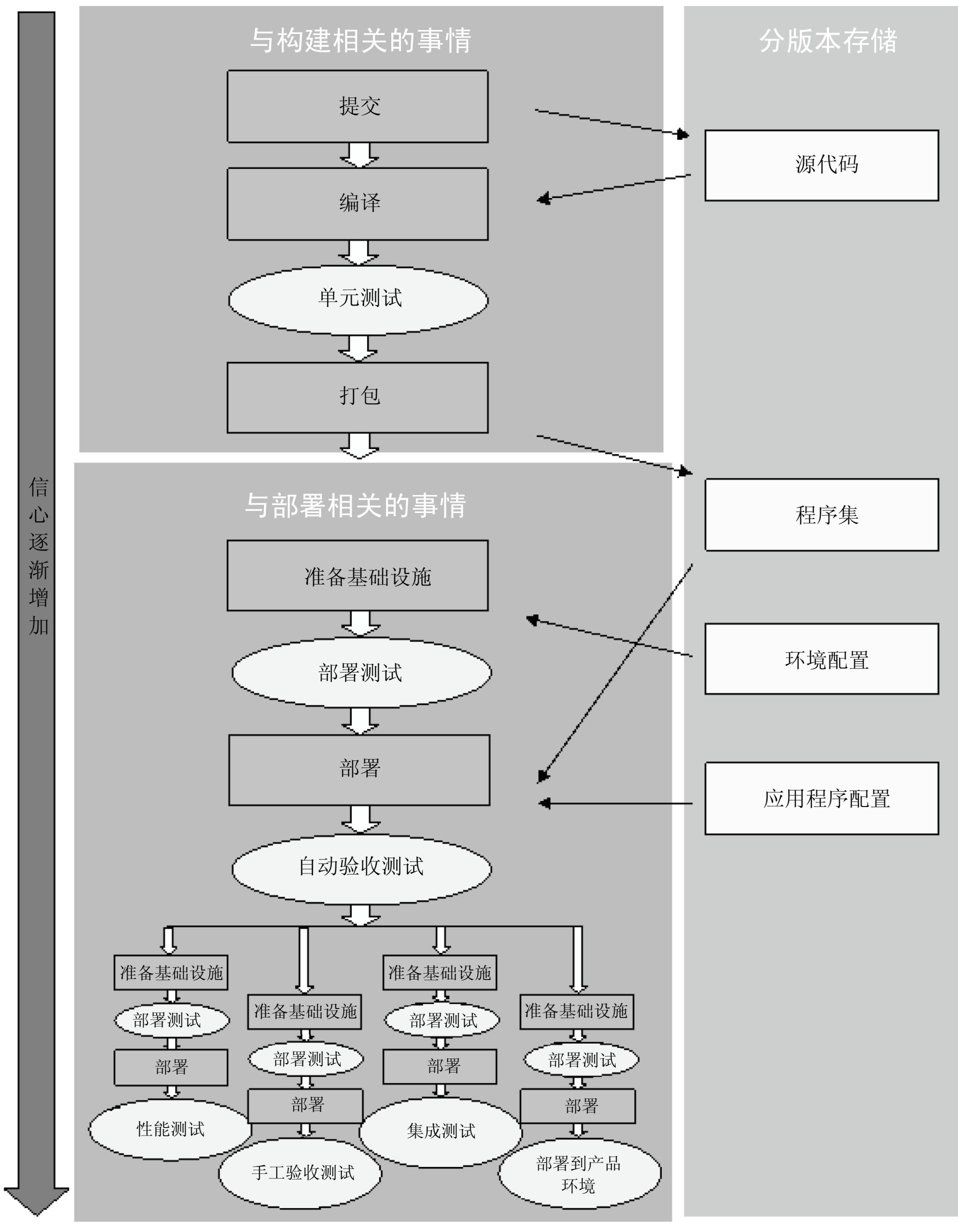


图12-1　持续集成管道

① 这里所说的“二进制文件”包括任何形式的经过编译的代码：.NET的assembly，Java的JAR、WAR和EAR，等等。重点在于：配置信息不包括在内。我们希望同样的二进制文件能在任何环境下运行。

二进制文件的管理

本方法的根本点在于：每通过管道中的一个阶段，该版本就向完全发布靠近一步。使用该方法最重要的理由就是：让错误走到最后发布的机会最小化。

假如我们只保存源代码，那么当以后需要部署某一版本时，我们还要重新编译源代码，假如我们打算用这个版本重新做一次性能测试，这次重新编译就有可能在性能测试环境中得到与前一次不同的结果。可能是因为本次编译使用了与前一次不同的编译器版本，也可能因为使用了不同版本的动态链接库等等。我们希望尽可能消除在部署后才发现那些本该在提交测试和功能测试时就该发现的错误的可能性。

“避免重复劳动”的指导思想会带来一些额外的好处：流程中每一步的脚本都倾于简单明了，而且鼓励将与环境相关的内容同与环境无关的内容分离。[①]

然而，当以这种方式进行二进制文件管理时，需要注意有一个问题，即这很容易浪费大量的存储空间去保存那些很少用到的二进制文件。一般来说，我们会把它们压缩后保存，而且尽量不将它们保存于版本控制系统中（因为这不值得）。我们使用共享文件系统做为二进制文件仓库，将二进制文件保存在打过版本标记并压缩过的镜像中。迄今为止，我们自己编写脚本来做这件事，因为还没有找到现成的替代方法。

这些镜像文件都标有与之相对应的源代码版本，而这种标识可以表明源代码、二进制文件及相应配置信息、脚本文件等所有相关信息之间的关联关系。

这些受控的二进制文件形成了近期构建的缓冲区，当容量达到一定值后，我们就会删除旧文件。一旦有某种原因使我们想将部署回滚到某一版本时，完全可以从该版本的标识中得到相关的源代码信息，并从代码仓库中找回源代码（但这种情况很少发生）。

12.4 第一道门——提交测试

这种自动构建管道由创建一个候选版本开始。而候选版本在任意一个开发人员向版本控制系统提交修改时产生。持续集成工具一旦发现开发人员的提交，立即编译代码并运行一组提交测试。这组提交测试包括所有的单元测试、一些冒烟测试，以及任何能够证明该版本质量满足发布候选条件的测试。

① 这一过程也隐含地反对“针对不同部署目标编译不同的二进制文件”的做法：这种与部署目标环境绑定的二进制文件无法灵活地部署。但在很多企业系统中这种情况仍然屡见不鲜。

这些提交测试的目标就是快速失败[①]。开发人员要等到这些测试全部成功以后才能做下一项任务。从这一点来说，速度是维持高效开发过程的关键。而在持续集成管道中，失败出现得越晚，其修复成本就越高。因此，出于维持团队高效性的目的，在这组提交测试套件中，除了单元测试以外，对其他测试应进行适当挑选。

一旦这组提交测试全部通过，虽然管道中后续的阶段还没有开始执行，但我们可以认为本次提交成功，开发人员就可以开始做下一个任务了。

这不只是过程级别上的优化而已。因为在理想情况下，如果所有的验收性测试、性能测试以及集成测试可以在很短的时间里完成，我们就不需要管道式持续集成了。 然而现实情况是：很多测试需要更长时间来执行，如果让整个团队停下来等着这些测试完全成功再继续工作的话，团队生产率无疑会显著下降。

将提交测试套件通过与否作为团队是否可以继续下一个任务的条件。当然，并不是说后续的阶段就不再关注了，而是希望团队时刻关注每次提交在管道中的执行情况，其目标是尽可能早地发现错误并将其修复，同时在长时间的测试执行过程中又可以让团队去做其他任务。

只有当这种提交测试的覆盖率足以捕获大多数错误的情况下，这种方法才可能被接受。如果大多数错误是被后续的阶段捕获的，那就表明需要加强一下提交测试。

我们希望开发人员能够尽早提交代码，可是这还取决于提交测试是否能够找到我们引入的绝大多数错误。要达到这一点，我们还需要通过不断试验来优化。

最初，我们可以将运行所有的单元测试作为提交测试，如果某个部分在后续阶段中经常失败，则再为其编写测试并将其加入到提交测试中。

12.5　第二道门——验收测试套件

单元测试是敏捷开发过程的关键部分，但仅有单元测试并不足以保证软件质量。全部单元测试都能通过的应用程序仍不满足业务需求的情况时有发生。

因此，除包括单元测试在内的提交测试之外，我所在的团队还依赖于自动化验收测试。这些

① 俗话说得好：早死早投生。——译者注

测试根据用户故事的验收条件编写，用于证明我们所写的代码满足用户验收条件。

这些测试就是一种对系统进行端到端验证的功能测试。如果所开发的软件与其他外部系统有交互的话，我们会清除外部连接点来完成这种端到端的功能测试。

我们将验收测试的创建和维护工作也纳入到开发流程中，即只有根据验收条件写好验收测试并将其加入到验收测试套件中、且通过了该测试，才能认为完成了一个用户故事的开发工作。我们还尝试让这些测试更加易读，让非技术人员也可以理解。然而这超出了本文所讨论的范围，就不在这里深究了。

验收测试运行于一个受控环境中，而且可以由持续集成管理系统（通常是CruiseControl）来监控。

在发布管理当中，验收测试是第二个关键点。我们的自动部署系统只会部署那些通过所有验收测试的软件版本。即任何版本都不可能绕过验收测试被部署到生产环境当中。

12.6 部署准备阶段

在某些项目中，可以实现部署自动化，即将应用程序自动部署到生产环境中；然而，对大型企业级应用程序来说，这基本上是不可能的。可是，如果我们能够将整个基础环境的管理和配置进行自动化，那会消除很多错误的来源，尤其是在企业级系统中手工操作如此之多的情况下更是如此。事情就是这样的，多少值得尝试一下，即便是只取得了部分成功，也能消除许多错误的来源。

我们采取了一些实用的方法来解决这一问题。例如我们通常会依靠标准服务器镜像、应用程序服务器、消息代理服务器以及数据库等。这些镜像表现为部署完整且包括基本配置信息的系统快照。

而这些镜像能以各种各样的形式存在，只要满足项目的需要即可。通常情况下，我们一定会有一个数据库脚本，用它来建立一个初始数据库结构并做一些数据初始化工作，还有一个标准操作系统或应用程序服务器的标准配置，它们可以作为委托过程的一部分被部署或建立在任何我们所选定的服务器上，而这个配置可能仅是一个文件系统的目录结构副本而已，所以总是具有相同的结构。

无论这些镜像是什么，其目的就是建立一个基础环境配置基线，以便以此为基础，对后续的变化进行维护，这样我们就迅速建立一个环境，而且会避免人工干预而产生错误。

但并不是每次部署软件时都重新建立这种初始环境。一般来说只在部署新环境时才使用，平时很少使用。但每次部署软件时，都会将其重置到基本点，以便让后续的部署工作建立在一个基线之上。

一旦这种基本环境准备好后，还要运行一些简单的部署测试。这些测试的目的在于确保当前的基础环境满足部署应用程序的基本要求，一般来说这些测试都非常简单，比如确保DBMS是存在的而且Web服务器可以响应请求等。

如果此时有测试失败的话，我们就会断定，要么我们的备份镜像有问题，要么我们的硬件环境有问题。

所有测试通过的话，我们就可以开始下一步的部署了。部署时，运行那些用于应用程序部署的脚本，它就会将已通过提交测试和验收测试的指定版本的二进制文件从二进制文件仓库复制到相应的路径去。

除拷贝文件之外，我们的脚本还能启动或停止应用程序服务器或Web服务器，运行脚本初始化或更新相应的数据库，可能还需要配置消息代理等。

基本上，部署过程共分为五步，其中四步是每次部署都需要的。

- 从镜像安装基础结构。（只有初始新的服务器环境时才需要做这一步。）
- 清理环境，使其重置到基本点，以便让后续的部署工作建立在一个基本点之上。
- 运行部署测试，以确保基础结构满足软件部署的基本要求。
- 将应用程序集放到相应位置。
- 对其进行适当配置，满足应用程序的运行需要。

我们将构建或部署脚本根据其责任分成多个部分，使它们尽可能的简单，而且每个脚本都仅完成一个具体任务，并定义尽可能清晰的输入参数。

12.7　后续的测试阶段

如前所述，验收测试是项目生命周期中的一个关键里程碑。某版本一旦通过了验收测试，便可以部署到各种不同的环境中，假如它没能通过这一步的话，也就不需要浪费人力去做部署。这表明了一个原则，即只有通过彻底的测试被证明可以正常工作的代码才能发布。

到验收测试为止，持续集成管道都是自动执行的。即新的版本通过前一步的测试后，自动进

入下一步，并运行该阶段的测试。

在大多数项目中，这种自动化方法在后续的步骤中就不那么适用了。因此，我们让后续的阶段成为可选项。即那些通过验收测试的版本可以选择性地进入人工验收测试阶段或性能测试阶段，亦或是直接部署到生产环境中。

对于这些阶段的部署工作来说，执行上面提到的部署过程五步骤有助于确保一个纯净的部署过程。当某版本被部署到生产环境时，这样的步骤应该已经被执行过几次，因此出现差错的机会很少。

我上一个项目中，每个装有我们的应用程序的机器上都有一个页面，用于显示候选的有效发布版本，还可以选择重新运行功能测试套件和/或性能测试套件。这为我们能够任意时刻无差错部署系统提供了高灵活性。

除了这些后续测试阶段的自动化程度可能有所不同外，整个过程基本是一致的。对于某些项目而言，可能有必要每次都运行性能测试，而对另外一些来说，可能就没有必要。其实，验收测试的后续阶段之间的关系以及自动化程度的高低并不是问题，而如何提供并管理这些持续集成过程的脚本才是需要考虑的事情。

12.8 让过程自动化

图12-2反映了自动化持续集成管道脚本。每个方框代表一个阶段，方框内的每一行代表了一个脚本来完成相应的功能。

大多数项目中，前两个阶段（提交测试及验收测试）通常可由诸如CruiseControl之类的持续集成工具来管理。

使用这种方法组织构建脚本的最大好处就是：每个脚本或元素都只负责做好一件事，而不是用相对复杂的步骤来管理整个构建过程。随着项目的演进与成熟，这一点对于确保构建过程的可管理性及易检验性是非常重要的。

至于如何写这些脚本并不在本文的讨论范围之内。实际上，这些脚本严重依赖于具体项目，不太可能引起读者的广泛兴趣。然而，当将这一过程应用于多个项目之后，我们发现它提供了可靠的、可重复的而且值得信任的部署过程，让我们可以在几秒或几分钟内完成此前可能需要数天才能完成的工作。

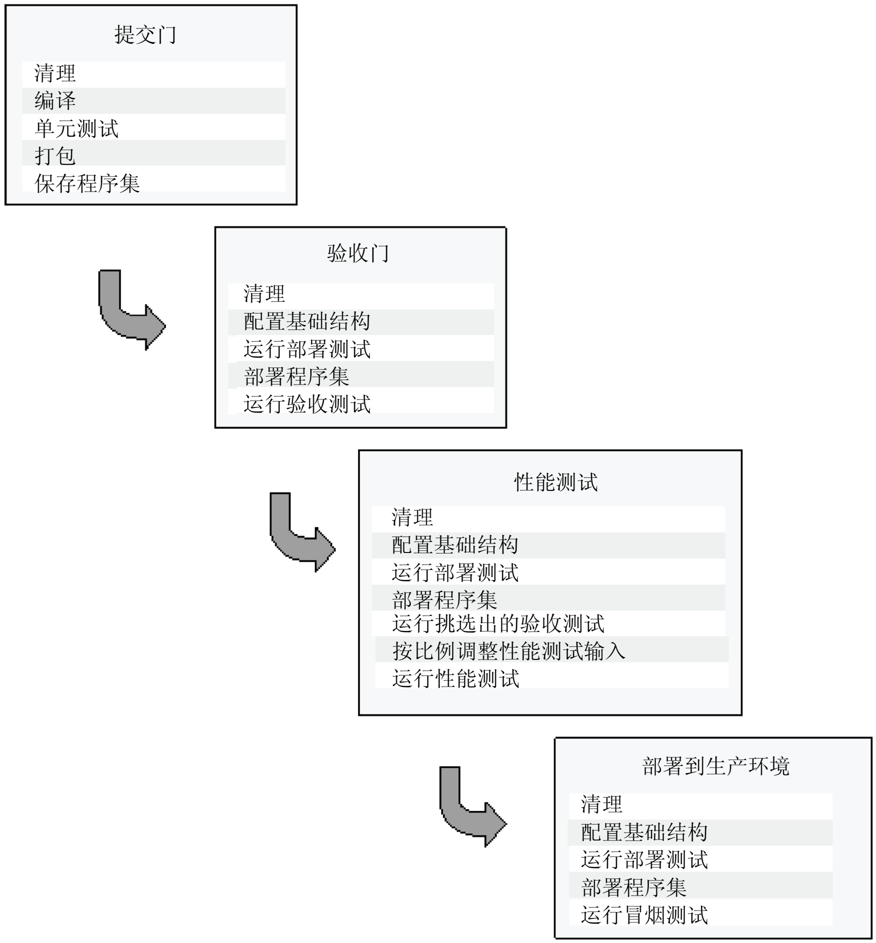


图12-2　过程示例

12.9　总结

如果你还没有使用持续集成方法进行构建，请从明天就开始吧。我们的实践证明，这是提高系统可靠性最有效的途径。持续集成在消除人为错误方面效果显著，不但可以提高生产效率，而且提高交付质量，降低交付压力。[①]

① 因成文较早，作者在项目中使用CruiseControl作为持续集成服务器，需要自编脚本去维护持续集成管道的配置，以及二进制文件仓库的管理工作。现在，Cruise（http://cruise.thoughtworks.com）已经可以轻松完成这些事情。——译者注

第 13 章

企业Web应用中的敏捷测试和瀑布测试

Kristan Vingrys，QA咨询师

13.1　简介

同是企业Web应用程序项目，一个用敏捷，一个用瀑布流程，它们的测试策略会有何不同？在二者中，测试的关注点都在于告诉业务客户这个应用程序做了哪些事情，同样也要消除应用程序作为产品交付以后的失败风险。它们的主要区别不是测试本身，而是何时执行测试、由谁执行测试。测试的每个阶段都可以在系统就绪后随时开始，无须等待前一个测试阶段完成。

从未涉足敏捷项目，或是刚启动某个敏捷项目并在寻找指导建议的读者都可以看看这篇文章，它正是为你们而写。文中的信息虽并非笔者新创，但亦是收集整理的结果，希望这些信息能帮助你们朝着更为敏捷的过程迈进。

敏捷项目的测试阶段跟瀑布项目的测试阶段几乎毫无二致。每个阶段都有准出标准，但是在进入某个测试阶段之前，是不需要等待整个应用程序完成的。只要应用程序所完成的部分足以进入下一个测试阶段就行。因为测试的对象是一个已完成的功能，而不是一个发布，所以测试阶段是在持续并行进行的。于是就有了一大堆回归测试。这便也意味着回归测试是测试自动化的基础。对于敏捷项目而言，环境与资源也是要注意的地方，因为对测试环境的需求会来得更早、更加频繁。

“快速失败”是敏捷项目的一句格言，它的含义是尽可能早地判断出应用程序无法满足业务需求。要做到这一点，就需要不断地对解决方案是否满足业务需求进行检测，一旦不满足，就要尽早把问题解决。敏捷团队成员——开发人员、测试人员、架构师、业务分析师以及业务代表等人都关注于尽早交付业务价值。所以，测试需要所有团队成员协力来做，它不仅仅是测试人员的责任。

13.2　测试生命周期

从测试生命周期就可以看出瀑布和敏捷项目之间最大的差异。瀑布项目对每个阶段都有严格的准入和准出标准，而且只有前一个阶段结束，才可以进入下一个阶段。而敏捷项目则会尽可能早的启动测试阶段，并且允许不同的阶段出现重叠。敏捷项目也有一些结构性的内容，如准出标准，但它没有严格的准入标准。

在图13-1中，你一眼就能看出敏捷项目和瀑布项目的测试生命周期差异所在。在敏捷项目中，业务分析师、测试分析师和业务代表等人一起讨论某个想法的行为，它如何适配于整体目标，怎样去验证它是否完成了该做的事情。这些分析构成了功能测试、用户验收测试和性能测试的基础。做完分析之后便开始功能的开发，单元测试、集成测试、探索性测试、非功能性测试（及数据验证——如果有这一项的话）也纷纷开始。不过只有等系统可以作为产品运行时才开始进行产品验证。

测试阶段没有严格的准入标准，这就意味着只要时机合适，随时都能开始。因为每个测试阶段对于确保应用程序的质量都至关重要，所以便应该尽早开始每个阶段的分析工作，这可以帮助人们修正设计，找出问题，为以后节省出大量的时间。下面是敏捷项目的一些准出标准。

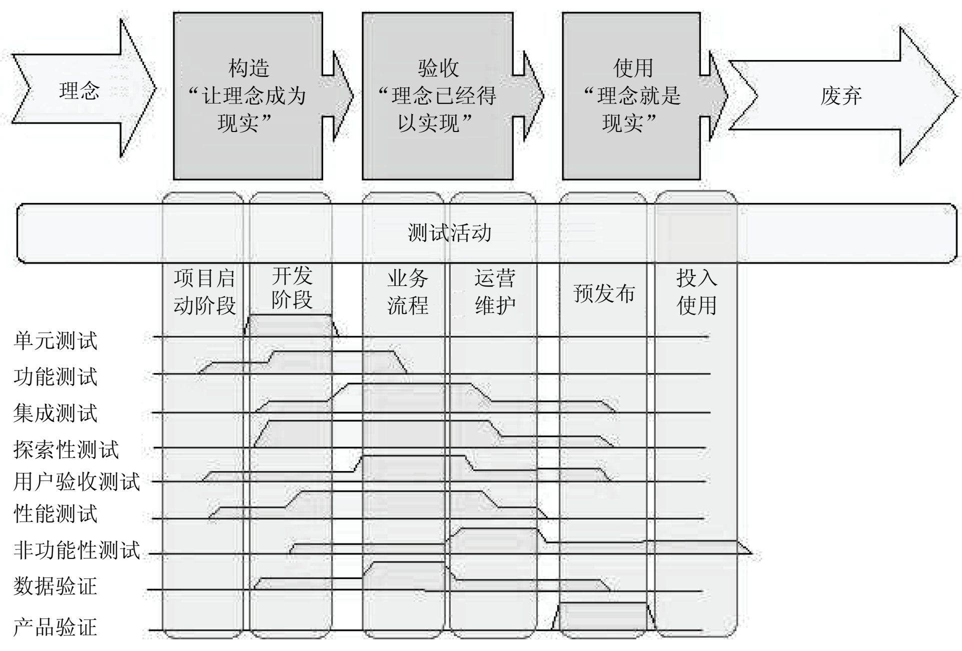


图13-1　敏捷项目和瀑布项目的测试生命周期

单元测试的标准：

- 100%自动化；
- 100%通过；
- 超过90%的代码覆盖率；
- 纳入持续构建。

集成测试的标准：

- 100%自动化；
- 100%通过；
- 纳入持续构建。

功能测试的标准：

- 90%以上自动化；
- 100%通过；
- 所有的自动化测试都纳入持续构建。

探索性测试的标准：

- 测试分析师对应用程序的质量有信心。

用户验收测试的标准：

- 业务代表认可应用程序满足需求；
- 用户认可应用程序的可用性。

性能测试的标准：

- 100%自动化；
- 业务人员认可应用程序满足了业务性能需求；
- 性能测试可以重复执行。

非功能测试的标准：

- 业务人员认可非功能需求得到了满足；
- 操作人员认可非功能需求得到了满足。

数据验证测试的标准：

- 确信数据被正确移植。

产品验证的标准：

- 应用程序在产品环境中正确安装。

瀑布项目的测试生命周期限制某个测试阶段只能在前面的测试阶段走完以后才能开始。从理论上讲这也是说得过去的，因为后面的测试阶段会依赖于前面的执行通过（如果某些功能不正确，就用不着再对它做性能测试了）。但是，要是非得等到所有功能都正确实现以后才开始性能测试，可就一点道理都没有了。敏捷项目会在适当的时候启动每一个测试阶段，这样可以尽早找出问题，让团队有更多的时间解决问题。但是敏捷项目的准出标准跟瀑布项目是一样的。除非功能都验证无误，否则就不能认为性能测试已经完成。

13.3　测试分类

敏捷项目跟瀑布项目的测试分类几乎是一样的。其差别主要在于大部分精力投入的地方和每个测试阶段所执行的时机。敏捷项目倾力关注单元测试和功能测试，从而为稍后执行的测试阶段创造出高质量的代码，于是后期就不会发现本应在早期发现的缺陷，并能专注于所需要测试的领域。而瀑布项目就有一个常见的问题，后期测试的焦点总是放在找出本来应该在前期被发现的缺陷身上。于是修复缺陷的成本提高了，测试工作的投入翻番了，测试的关注点也偏离了。

瀑布项目和敏捷项目另外一个巨大的不同在于测试自动化。敏捷项目力求在所有测试领域内都达到100%自动化。测试跟持续构建系统集成在一起，于是当代码发生变化时，这个变化会被自动检测到，应用程序被构建起来，然后所有测试都会被执行。

测试驱动开发（TDD）在敏捷项目中很常用，用这种方法的时候，测试用例比代码要先一步创建。于是我们就可以越来越多地看到为代码和功能创建的测试用例。用自动化测试来驱动开发，然后消除重复，这种做法可以保证无论复杂度多高，任何开发者都可以写出可靠的、没有bug的代码。TDD常用的地方是单元测试，但也同样可以用于功能测试、集成测试、用户验收测试和性能测试。

单元测试

单元测试又称白盒测试，它要测试所开发出来的每一个模块。瀑布项目不关注这个测试阶段，而且大多数时候即便有的话也是随意为之。敏捷项目强调单元测试，而且会把所有单元测试都自动化。自动化的单元测试是敏捷项目的基础，对持续集成和重构起辅助作用。

单元测试应当考虑以下几点：

- 用stub和mock来消除对外部接口的依赖；
- 由创建代码的开发人员编写单元测试；
- 单元测试要能自动化执行，而且要被包含在持续开发构建中；
- 单元测试之间不能有依赖，让每一个单元测试都能独立执行；
- 任何开发人员都要能在他自己的机器上执行单元测试；
- 用代码覆盖率来判断哪些部分的代码没有被单元测试覆盖；
- 在检入代码的修改之前，要保证单元测试100%通过。

任何测试失败都表示构建失败。

功能测试

功能测试常常跟系统测试相关联，它的重点是测试应用程序的功能（包括负面测试和边界条件）。在瀑布项目中，测试团队一般是在这个阶段开始测试工作。测试团队的成员会一直等下去，直到开发者完成了所有的功能，并通过所有单元测试之后才进入功能测试阶段。敏捷项目把功能拆分成故事，每次迭代开发一定数量的故事。每个故事都有一些验收标准，这些标准一般都是由业务分析师和测试分析师制定的，它们也可以看作跟测试条件类似。测试分析师会根据验收标准创建测试用例，用它们来检测代码行为的完成程度。故事一旦编码完成，而且单元测试运行通过以后，就可以运行功能测试来判断是否满足验收标准了。这就意味着在敏捷项目中，只要第一块功能已经完成编码，功能测试就可以启动，而且会贯穿整个项目的生命周期。

功能测试应当考虑以下几点。

- 要能自动化执行，并且进入持续构建（如果测试运行时间很长，也可以只在开发持续构建中包含一小部分精挑细选的功能测试，而在系统集成持续构建中包含全部功能测试）。
- 在编码之前写下测试意图，代码完成后对测试进行实现。
- 把通过所有功能测试作为故事完成的条件。
- 在应用程序安装到另外一个环境（如试机环境或产品环境）上的时候需要执行功能测试。

任何测试失败都表示构建失败。

探索性测试

探索性测试又称随机测试。瀑布项目在它们的测试策略中没有这种测试类型，不过大多数测

试人员都会多少做一些探索性测试。在敏捷项目中这是个很关键的测试阶段，因为它可以用来检测自动化测试的覆盖率，并收集对于应用程序质量的反馈。它为测试人员和业务分析师提供了一种结构化的方式去使用、去探索系统，从而找到缺陷。如果探索性测试在某个功能区域内找到了大量缺陷，那这部分的自动化测试用例就要被审核。

探索性测试应当考虑以下几点：

- 在系统集成环境中执行；
- 在较高的层次（如在wiki中）上监测探索性测试活动；
- 用自动化的环境设置方式来减少搭建环境的时间；
- 将破坏性测试作为探索性测试的一部分。

集成测试

应用程序在作为产品部署的时候，需要各个部分的协作；集成测试就是把这各个独立的部分集成起来进行测试。瀑布项目不但会把应用程序的各个独立模块进行集成，还会把那些虽然不属于项目的一部分，但是要开发当前应用程序就必须用到的一些应用程序也做集成。在敏捷项目中，独立模块间的集成是被持续构建所覆盖的，所以集成测试的关注点就是那些不属于当前项目的外部接口。

集成测试应当考虑以下几点。

- 在执行集成测试的时候，需要考虑到当前迭代还没有开发的功能；
- 写一些集成测试来定位特殊的集成点，以协助代码调试，即便功能测试会调用到这些集成点也无妨；
- 将集成测试自动化，放到系统集成持续构建中。

任何测试失败都表示构建失败。

数据验证

如果项目需要移植既有数据的话，就要进行数据验证。它可以保证现有的数据被正确地移植到新的Schema中，新的数据被添加进来，旧的数据被移除。这种类型的测试在瀑布项目和敏捷项目中是被同等对待的，除了敏捷项目会尽力把它自动化以外。

数据验证应该考虑以下几点：

- 在系统集成环境、试机环境和产品环境中都要执行；
- 尽可能地自动化；
- 要把测试放到系统集成持续构建中。

任何测试失败都表示构建失败。

用户验收测试（UAT）

UAT关注的是完整的业务过程，它用来确保应用程序能按照业务的处理方式工作并能满足业务需求。它同样还要从客户、消费者、管理员等各种用户的角度出发考虑应用程序的可用性。在瀑布项目中，这个阶段通常就被用来找出应该在早期阶段就被发现的bug。业务人员也会在这个阶段验证开发团队交付的应用程序的质量。敏捷项目用UAT来确保应用程序满足业务需求，因为等到进入这个测试阶段的时候，代码质量已经较高了。在敏捷项目中，业务人员从早期的测试阶段就开始参与，所以他们对交付的东西有更多的信心。

UAT应该考虑以下几点：

- 首先进行手工测试，等它验证了系统行为以后再把它自动化；
- 把自动化测试放到系统集成持续构建中；
- 让应用程序的最终用户亲自将整个程序运行一遍，不过项目的测试人员要在旁边协助；
- 在试机环境下执行UAT，用作验收；
- 只要完成了一个业务过程或者一个主要的UI组件，就要执行UAT。

任何测试失败都表示构建失败。

性能测试

性能测试涵盖的范围比较大，不过一般可以分成以下三类。

- 容量测试：独立测试核心功能组件的容量。例如，可以支持多少用户并发搜索？一秒钟能做多少次搜索？等等。容量测试被用来精确度量系统的极限，还可以对容量规划和系统的扩展性起辅助作用。
- 负载测试：侧重于系统在负载下的表现。负载应该要体现出用户所期望系统可以经受得住的流量。
- 压力测试：关注系统在压力下的表现。比较常用的技术是浸泡测试（soak test）；在浸泡测试中，系统会在一定的负载下持续运行一段时间，用来找出长期问题，如内存泄漏、

资源泄漏等。压力测试还会覆盖到故障转移和恢复，例如让正在工作的集群中的一台服务器出现问题，检查是否可以做到故障转移和恢复。

瀑布项目直到项目接近尾声的时候才做性能测试，这个时候应用程序已经“完成了”开发，通过了单元测试和功能测试。而敏捷项目则会尽快启动性能测试。

性能测试应该考虑以下几点。

- 在功能测试中设置一些性能度量，例如一个测试第一次运行要花多长时间，接下来每次又要花多久，把这些时间所占的百分比做一下比较（上升表示有问题，下降表示良好）。
- 把一些性能测试放到系统集成持续构建中去做。
- 一旦一个业务过程，或是某个规模比较大的功能或接口完工，就要做性能测试。
- 只有在试机环境中运行的时候才签收性能测试。

任何测试失败都表示构建失败。

非功能性测试

非功能性测试所涵盖的范围很广，性能测试通常也属于这个话题。但是因为性能测试是企业解决方案中很重要的一部分，而且需要不同的资源和技能集，所以就被划出来单独成为一个测试阶段。非功能性测试一般都包含有这些内容：可操作性（包括监控、日志、审计/历史记录）、可靠性（包括故障转移，单个组件故障，完整故障，接口故障）以及安全性。无论瀑布项目还是敏捷项目，在这个测试阶段都会遇到重重阻碍，二者的差异不太突出。

非功能性测试应该考虑以下几点：

- 非功能性需求一般都是很难捕获的，而且即便被捕获，也很难进行度量（例如，99.9%的无故障时间）；
- 尽可能让所有的非功能测试都自动化执行，把它们也都放到系统集成测试环境中；
- 在定义测试用例的时候，让真正要监控产品环境并对其提供支持的人也参与进来；
- 在应用程序成为产品以后，非功能测试或监控还要继续。

回归测试

在瀑布项目中，回归测试无论从时间上还是费用上来看，都算得上是成本最高的测试阶段了。如果在项目晚期（如UAT阶段）发现了缺陷，再构建应用程序的时候，就得把所有单元测试、功能测试和用户验收测试重新运行一遍。因为大多数瀑布项目都没有自动化测试，所以回

归测试的开销就很大。敏捷项目以持续构建和自动化测试来应对回归测试，使每次构建都可以进行回归测试。

回归测试应该考虑这一点：

- 每次迭代结束时都做一下手工测试（如果规模很大的话，就进行拆分，做到每三四次迭代就能执行完一次），收集早期反馈。

产品校验

产品校验会在把产品交付给用户使用之前，审查产品环境中的应用程序，看看所有内容是否都已经正确安装，系统的操作性如何。而有些测试还是只能到了产品环境中才能完整运行的，最好尽快将其完成。无论是瀑布项目还是敏捷项目，产品校验的方式并无二致。

产品校验应该考虑以下几点：

- 让最终用户来做产品校验测试；
- 趁着产品系统还没有正式上线，从这个测试阶段的早期就要尽可能多地执行自动化回归测试。

瀑布项目和敏捷项目的测试阶段还是很相似的，主要差异就是每个测试阶段所关注的重点和执行时机。敏捷项目中有大量的自动化测试，用持续集成来减少回归测试对项目带来的影响。在瀑布项目中，如果发现应用程序的质量低下，那么在晚期再去执行前期的测试就是很常见的事情（如在UAT的时候作功能测试）。敏捷项目可以减少测试中的浪费，在早期发现问题，让团队在交付应用时增强信心。

13.4　环境

在开发过程的各个阶段都要用到测试环境，从而确保应用程序的正常运行。越到后期，测试环境与预期的产品环境就会越相似。测试环境一般都会包括一个开发环境，让开发者集成代码并运行一些测试。系统集成环境跟开发环境有些类似，不过它会集成更多的第三方应用程序，也许还有大批量的数据。试机环境几乎是产品环境的镜像，也是应用程序变成产品之前的最后一个阶段。

敏捷项目和瀑布项目所需的环境没太大区别，其中一个不同之处在于，敏捷项目从项目伊始直至项目结束，都要用到所有的环境。在敏捷项目中，保证所有的环境都一直正常工作也是很重

要的。无论因为什么原因让某个环境出现故障，都要立刻让它重新工作起来。在这个话题上，敏捷和瀑布还有另外一点差异，那就是环境的计划和资源分配对它们的影响不同，尤其是当各种环境都被项目之外的团队进行管理的时候，其差异尤为显著。

开发集成环境

开发者在开发环境中集成代码，开发应用程序。瀑布项目对开发环境的重要性不会考虑太多：开发环境中的代码一直都不能工作，到了开发者需要彼此集成代码的时候才想起来要用，而这时项目已经临近尾声。在敏捷项目中，开发环境是整个开发工作中不可分割的一部分，在开始编码之前就必须准备就绪。这个环境会被用来持续地集成代码和运行测试套件。无论因为什么原因造成环境故障，都要立刻修复。

开发环境应该考虑以下几点。

- 集成代码、构建和测试的时间加起来不要超过15分钟。
- 每个开发人员所用的环境跟开发环境要保持一致（硬件环境可以不一样，但是软件环境一定要一样）。
- 所用的数据要尽可能跟产品数据保持一致。如果产品数据过于庞大，难以载入，也可以只截取一部分。在每个发布周期的开始阶段，这些数据要从产品数据中重新更新。
- 管理这个环境的责任应该落在项目团队身上。
- 向这个环境中部署的频率大约是以小时计算。
- 自动部署。

系统集成环境

系统集成环境用来将所开发的应用程序和其他应用程序进行集成。在瀑布项目中，这个环境（如果有的话）只会在项目临近尾声的时候才会用到，而且倾向于多个项目共用一个集成环境。在敏捷项目中，一旦开始编码，这个环境就要准备就绪。应用程序会被频繁部署到这个环境中，继而开始执行功能测试、集成测试、可用性测试和探索性测试等等。应用程序的演示就是在这个环境中进行。

系统集成环境应该考虑以下几点。

- 集成点应该被真正的外部应用程序所代替。外部应用程序应该处于测试状态，而非真正的生产版本。
- 把产品环境的架构复制过来。

- 在这个环境中所使用的数据应该是产品环境数据的副本，每个发布周期的开始阶段，都要从产品数据中重新更新。
- 建立运行这个环境中所有测试的系统集成持续构建。
- 管理这个环境的责任应该落在项目团队身上。
- 向这个环境中部署构建的频率大约是以天计算。
- 自动部署应用程序。

试机环境

试机环境用来验证应用程序可以部署为产品，而且工作正常。为了达到这个目的，试机环境应该完全复制产品环境，包括网络配置、路由器、交换机以及计算机性能等等。在瀑布项目中，这个环境往往需要“预订”，也要有一个计划，计划在这个环境中进行多少次部署以及何时进行部署。敏捷项目对这个环境不像对开发环境和集成环境那样依赖；不过在项目的整个生命周期中，还是需要常常进行部署。

试机环境应该考虑以下几点。

- 这里的数据应该是产品数据的完整副本，每次应用程序部署前都要更新。
- 用它来验收UAT，性能测试和非功能测试（稳定性、可靠性等等）。
- 向这个环境中部署构建的频率大约是以迭代计算 ，如每两周一次。
- 管理这个环境的人应该就是管理产品环境的人，让他提前接触应用程序并进行知识传递。
- 自动部署应用程序。

产品环境

产品环境是应用程序正式上线时的环境。产品校验测试就在这个环境中执行，同时会做一些度量以检验测试成果。瀑布项目和敏捷项目在这里的区别不大。在敏捷项目中，发布过程会尽力做到自动化，为向产品环境中频繁发布提供条件。

产品环境应该考虑以下几点。

- 在应用程序正式上线之前（或者刚刚上线之后），在产品环境中做回归测试。
- 要做度量，以检验测试成果，例如在前三个月到六个月之前，用户报告了多少问题，问题的严重性如何。

无论是哪种项目，要保证时间期限，就都得做到在需要用到环境的时候就有一个环境可供使

13

用。瀑布项目会设法遵守严格的计划，便于对环境做安排。敏捷项目的灵活性更强。也许有了足够的功能就可以进入试机环境了，或者业务人员会决定产品提前上线。为了做到向试机环境快速部署，然后进入产品环境，就要有系统集成环境以及相关过程的辅助。

13.5　问题管理

问题包括缺陷（bug）和变更请求。瀑布项目有着严格的缺陷和变更请求管理，而敏捷项目饱含变化，所以就没有那么严格的变更管理。如果需要有变更，就创建一个（或是一些）故事，放到待处理事项里面。高优先级的故事会放到下一次迭代中完成。

缺陷管理在敏捷项目中依然适用。如果有人发现一个正在开发故事有缺陷，大家就会进行非正式的沟通，发现缺陷的人把它告诉开发人员，缺陷会立刻被修复。如果某个缺陷并不属于当前迭代中开发的故事，或者属于当前故事，但并不严重，那就用缺陷跟踪工具记录下来。这个缺陷会被当做故事处理；也就是说，会创建一张故事卡，让客户排优先级，放到待处理故事里面。团队需要进行权衡：要找到问题所在，为大家理解这个问题并安排优先级提供足够的信息，但又不能在一个对客户而言优先级并不是很高的缺陷上面花太多时间。

瀑布项目和敏捷项目的缺陷内容（描述、组件、严重程度等）是一样的，除了一个字段以外：敏捷项目多了一个字段——业务价值，如果可能的话就用币值描述。这个字段表示如果缺陷被解决的话可以带来多少业务价值。将业务价值跟缺陷关联以后，客户就更好地判断这个缺陷跟新功能相比是不是更有价值，是不是应该有更高的优先级。

13.6　工具

所有项目都要用到工具，只是程度不同。瀑布项目用工具来强化过程以及提高效率，这有时会造成冲突。敏捷项目用工具辅助提升效率，与过程无关。在敏捷项目中，所有的测试都应该可以在任何团队成员的个人环境中运行，也就是说，所有人都可以使用那些自动化测试用例的工具。所以敏捷项目中会经常用到开源产品，这又意味着使用这些工具需要不同的技能。开源工具不像商业工具那样有齐备的文档和完善的支持，用这些工具的人要有很强的编码能力。如果有人编程能力偏弱，就可以通过跟人结对来提升个人技能。在敏捷项目中也可以使用商业工具，但是大多数商业工具在开发的时候都没有考虑敏捷过程，所以跟敏捷项目匹配起来就不太容易。而且要让商业工具跟持续集成配合，可能要写很多代码才行。

项目中应该考虑为下面一些测试任务使用工具：

- 持续集成（如CruiseControl，Tinderbox）；
- 单元测试（如JUnit，NUnit）；
- 代码覆盖率（如Clover，PureCoverage）；
- 功能测试（如 HttpUnit，Selenium，Quick Test Professional）；
- 用户验收测试（如Fitness，Quick Test Professional）；
- 性能测试（如JMeter，LoadRunner）；
- 问题跟踪（如BugZilla，JIRA）；
- 测试管理（如Quality Center）。

13.7　报表与度量

度量数据是用来衡量软件质量和测试成果的。在瀑布项目中，有些测试度量指标需要在测试之前就把所有测试用例都写好，而且仅在应用程序开发完毕时进行一次测试。这种指标包括：每个测试用例执行的时候发现多少缺陷，每天执行的测试用例会发现多少缺陷。这些度量数据收集起来以后，便用来判断应用程序是否已经就绪并可以发布。在敏捷项目中，功能完成的时候测试用例就已经写好且运行完毕，这就意味着用来度量瀑布项目的一些指标是无法应用在这上面的。

回到收集度量数据的原因上来——衡量软件质量和测试成果，你可以看看下面这些概念。

- 用代码覆盖率度量测试效果；这对于单元测试尤其有效。
- 在探索性测试阶段发现的缺陷数量可以说明单元测试和功能测试的效果。
- 在UAT阶段发现的缺陷表示先期的测试并不像UAT一样充分，我们应该关注业务过程，而不是软件的bug。如果UAT发现了很多功能性问题，而不是软件的bug，这就表示团队对故事或是变化的需求理解不足。
- 故事完成以后所发现的缺陷数量能够作为衡量软件质量的好手段。这些缺陷包括在集成测试、非功能测试、性能测试和UAT测试中发现的缺陷。
- 缺陷重现率。如果缺陷常常重现，软件质量就很低。

13.8　测试角色

测试角色并不是跟单个资源一一对应的。一个资源可以担任多个测试角色，一个测试角色也可以由多个资源负责。下面列出的这些角色是确保项目测试效果所必需的。一个优秀的测试人员

应该具备所有这些角色的特征。敏捷项目和瀑布项目都有这些角色，只是扮演这些角色的人不同。在敏捷项目中，所有团队成员都会扮演一些测试角色，在图13-2中展示了一个例子，你可以看到在敏捷项目中，每个团队成员都是怎样扮演各个角色的。这并不是强制性的规定；每个团队各有差异，不过这种做法也算得上是不错的组合。

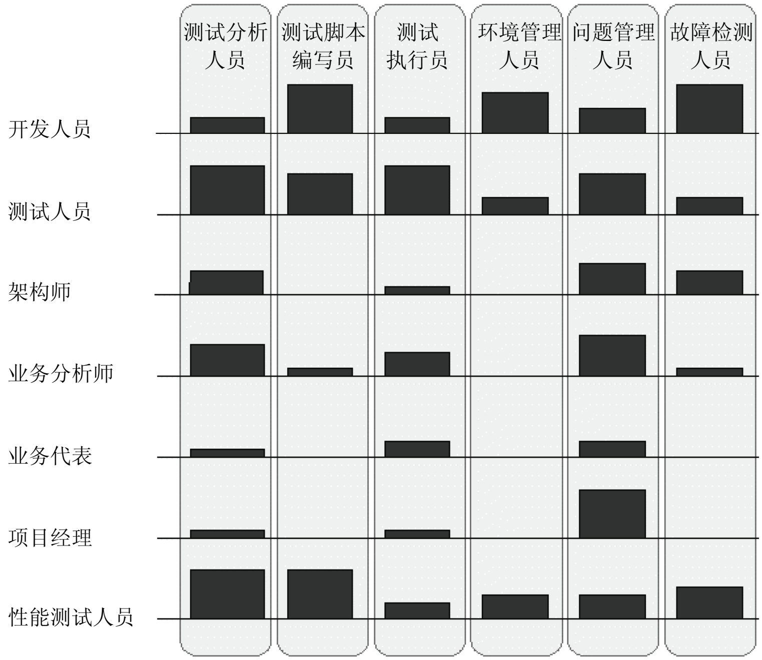


图13-2 不同团队成员的测试角色

测试分析人员

测试分析人员要了解需求、架构、代码等各个产物，从而判断哪些需要做测试，哪些是测试要重点关注的地方。

在瀑布项目中一般是有一个（或多个）资深的资源来担任这个角色。他们检查相关文档（需求、设计、架构），编写测试计划，编写高层的测试用例描述，然后把所有的东西都交给一个初级员工，让他填补详细的测试脚本。敏捷项目鼓励所有团队成员一起担任这个角色。开发人员的关注点是通过分析代码和设计来编写单元测试，但是他们也会协助业务分析师或者测试人员编写功能测试，还会参与非功能测试和性能测试的分析。业务分析师和测试人员紧密协作，编写功能测试和用户验收测试，并执行探索性测试。客户/最终用户会被邀请参与用户验收测试。

测试脚本编写员

该角色就是编写详细测试脚本的人。这些脚本可能供手工执行，也可能被自动化。瀑布项目中的脚本编写员就是个初级员工，他根据测试计划和测试用例描述来编写手册，每一步都描述的很详尽。自动化脚本编写员就得是更资深的人了，开发人员也会参与单元测试用例的编写。敏捷项目会大量使用开发人员来编写测试脚本，主要是因为测试脚本是自动化执行的。

测试执行员

不管是手工测试还是自动化测试都有这个角色，不过在自动化的时候，这个角色的扮演者就是一台电脑。测试执行员会执行详细测试脚本，判断哪些测试失败，哪些测试通过。瀑布项目一般都用测试人员来做这件事情，而敏捷项目则鼓励所有人都来参与，尤其是测试人员、业务分析师和客户。

环境管理人员

这个角色管理测试环境，包括应用程序运行的环境以及支持自动化测试的基础架构。他们还会关注外部接口和用作测试的数据。这个角色在瀑布项目和敏捷项目中很相似。

问题管理人员

问题出现以后就要解决。这个角色可以帮助筛查问题，确保它们被正确地创建，有各种属性，如严重程度、优先级、组件等等。这个角色还要管理问题的生命周期，并提供工具支持。这个角色在瀑布项目和敏捷项目中很相似。

故障检测人员

这个角色当问题出现的时候就要去做故障检测工作，判断是不是软件缺陷。如果是软件缺陷，他们就要去找出问题根源、可能的解决方案和变通措施。这个角色在瀑布项目和敏捷项目中很相似。

敏捷团队所注重的是让各个角色得到充分发挥，而比较少关心谁在做什么事情、谁对哪些事情负责。测试人员和其他团队成员之间没有界限，他们共同的目标是生产出更高质量的软件，每个成员都要尽一切可能帮助达成这个目标。在敏捷团队中，测试人员可以从所有人那里得到帮助，而他们又可以帮助其他人提高测试技能。这种方式能够确保团队中的每个人都在为交付高质量应

用程序而付出。

13.9　参考文献

Test-Driven Development：*By Example* by Kent Beck（Addison-Wesley Professional，2002）

“Exploratory Testing Explained v.1.3 4/16/03，” copyright 2002–2003 by James Bach，http://www.James@satisfice.com

第 14 章

实用主义的性能测试

James Bull，QA咨询师

性能不彰的软件不仅不能让人们的生活变得轻松，反而会妨碍企业的正常运转，并且让使用它的人心烦。无论实现了多少功能，这样的软件都只能让用户感觉质量很糟糕。一旦你的软件给用户留下了糟糕的体验，他们不会等到你下一次的改进，而会决定掏钱去购买别人的软件。

假如一个系统又快又可靠，伸缩性又好，而另一个在这几方面表现都不好的话，用户当然会选前者。我们这里所说的“性能测试”不仅针对性能，而且通常还包括对伸缩性和可靠性的测试，因为这些测试往往可以同时、用同一套工具来完成。在这篇文章里，我将会介绍如何确保最终产品能够具备这些好的属性——我通常把它们统称为性能。

14.1　什么是性能测试

大家应该都能同意：良好的性能不只是好东西，更是一个值得为其花钱的东西。现在的问题是：应该在哪里做性能测试？如何让性能测试帮助我们写出性能良好的软件呢？

性能测试应该囊括确保产品性能符合要求所需的一切行动。这里有四个关键点：需求、产品性能数据、沟通和流程。

这四点中一旦有所缺失，性能测试的效果就会大打折扣。假如仅仅做测试，情况并不会真正有所改观，因为你并不知道系统应该有多快。所以，你需要采集需求。假如测试的结果没有得到有效沟通，那么就没人知道问题的存在，也就不会采取任何行动来解决问题。即便我们采集到了需求，对产品做了测试，也把结果告知相关人员，工作仍旧没有结束。假如项目计划中没有给“解决性能问题”留出空间，或者没有一套流程根据测试结果计划后续的工作，那么你还是没办法对

最终交付的软件产生太多影响——在这种情况下，你耗费大量成本做性能测试，结果只是让你知道产品的性能究竟有多糟糕或多强大，却对这个结果束手无策。

我们想要的不仅仅是知道结果，更希望所获得的信息能对我们正在开发的软件产生影响，从而保证我们能够满足——或者至少接近——用户对性能的要求。下面我将逐一讨论这四个关键点。

14.2　需求采集

性能需求采集的重要性经常被人们低估。在这一节里，我将尝试阐明几个重要问题：要度量什么？如何知道我们需要什么？以及如何得到确实有用（而非帮倒忙）的数据？

要度量什么

最重要的性能度量点有两个，最大吞吐量以及给定吞吐量下的响应时间。一个好的做法是：分别度量几种不同吞吐量下的响应时间，从中分析负载对响应时间的影响。如果响应的及时性非常重要，那么在确保满足响应时间要求的前提下所能达到的吞吐量可能就会明显低于最大吞吐量。你需要通过度量找出两项数据：当响应时间恰好可以接受时的吞吐量，以及达到预期吞吐量时的响应时间。伸缩性度量的关键则在于：随着数据规模、用户数量或者运行系统的硬件变化，起初得到的性能度量数据会发生怎样的变化。

可靠性的关键度量点是：当负载量高得超乎寻常，或者连续运行了很长时间以后，系统是否仍然正常工作。

如何设定目标

要想知道系统需要达到怎样的吞吐量，你首先需要知道有多少用户会使用这个系统，以及他们的使用模式。用户会多频繁地使用某个功能？这个功能需要多快完成？

业务用户会知道这些问题的答案。你应该让他们明白，你会经常需要向他们咨询这方面的事。然后你应该建立一个良好的沟通流程，以确保信息的获取畅通无阻。

总而言之，你需要有一个可靠的流程与机制来获得所需的信息，及时获知支撑业务需求所需的性能指标。如果不经常去计算这些数据，就有可能发现你正在朝着已经过时的目标努力。

弄清当前需要负载的吞吐量之后，下一个需要考虑的就是响应时间。在结合UI考虑这个问题

时，人们常会有钻牛角尖的想法。既然用户界面要在几秒钟之内响应，那么功能自然必须在更短的时间内完成。但事实并非如此。UI应该立即响应，告知用户：他们的请求已经得到处理；但实际的处理未必马上完成。在整个过程中，系统的其他部分应该照常工作。

响应时间的目标应该主要针对用户界面，并且数值越低越好。而且，不应该期望所有功能都能在同样的一段时间内完成。

如果对前面所说的还不明白，下面我将简单介绍一个采集性能需求的流程。

如何将性能测试融入日常开发流程

理想情况下，项目组每周应该召开一次会议，确定当前的性能需求。参加这次会议的人应该包括项目经理、关注性能的客户、资深开发者、以及性能测试人员。如果某些性能需求明显无法达到或者完全不合理，开发者需要在第一时间指出。客户的参与是为了提供业务上的信息与知识，从而帮助判断需求的合理性。项目经理需要知道团队做了哪些决定，并提供一些方向性的指导。至于性能测试人员，他们显然应该在场，这样他们才知道需要测试什么。

接下来，你需要找到适当的讨论对象。开发团队需要从客户中找到一个联系人，与他一道决定性能需求，这样才能确保客户和开发者都清楚目标所在。不要把性能需求看作神圣不可侵犯之物，和所有需求一样，它们也应该是开发者与客户对话的起点，双方需要共同讨论决定最终的目标。

一旦需求确定下来，就能决定当需求得到满足时如何向客户展示，并跟其他的任务一样对编写测试的工作进行评估和计划。

开发者需要性能测试告诉他们什么

开发者的需求有很多种，但背后的驱动力总是一致的。如果某段代码需要返工，他们就需要更多的信息来了解当时的情况。这些信息可能来自代码检查工具，也可能来自线程转储，甚至来自日志。他们可能需要知道与应用程序服务器相比数据库的忙碌程度，或是负载达到峰值时网络的忙碌程度。

预先回答所有这些问题可能并不值得一试，因为这会需要很大工作量。我们能做的是：当问题出现时，弄清哪些信息会有助于开发者解决问题，然后把获取这些信息的任务加到你的任务列表上，并告知客户。此时你就可以考虑以下问题：从此刻开始为所有测试获取信息是否容易，这是否是针对眼下的特定问题所做的一次性测试。

如果开发者的需求是以这种方式在会议上提出的，那么，所有人都会知道这些需求的存在。客户可以为这些需求排优先级，可以把它们纳入项目计划。最终性能测试将满足各方的需求：它让客户对正在开发的软件保持信心，它也能帮助开发者找到并解决性能问题。

找不到关注性能的客户怎么办

如果找不到一个关注性能需求的客户，就会有以下风险。首先，正在开发的软件可能不符合业务要求，项目可能彻底失败。其次，不管最终的产品如何，客户都可能说它不符合要求，因为他们感觉开发团队没有征求他们的意见。最后，这可能会在团队内部造成紧张气氛，开发团队会觉得自己在被迫做不必要的工作，因为需求不是来自客户——不管项目经理的担心是否正确，这种想法都有可能出现，并导致必要的工作没有被完成。亦或相反，开发者们浪费时间去做不必要的工作。

如果客户不懂技术又非要坚持不可能的需求该怎么办

这种可能性总是存在：客户希望产品的性能达到某个水平，而达到这个水平是不可能或者不经济的。这时你就需要提出一些中肯的问题，把对话引导到真实的业务需求上来，从而打消客户不切实际的要求。

如果客户的要求是关于吞吐量的，可以考虑的问题有：每个工作日处理多少事务？这些事务的时间分布如何？是平均分布还是有明显的峰谷之分？每个周五下午会有集中访问，还是说峰值的出现没有特别的模式可循？

关于响应时间，可以考虑的问题有：用户界面的响应时间会对系统的处理能力造成什么影响？能不能把界面与实际的计算操作分离？比如说，可能有这样一种场景，用户输入一些数据，然后进行较长时间的数据处理。此时用户不希望一直等到处理完成，而是希望立即输入下一段数据。所以这时合理的期望不是在一秒钟内完成数据处理，而是将用户界面与数据处理分离，让系统在后台处理前一段数据，同时让用户在界面上输入更多的数据。

通过上述方式，我们就能让开发者和客户共同寻找一个对业务价值有意义的性能水平，并且分清什么是当务之急以及什么是锦上添花。我们都曾遇到下面这种情况：在项目的现有条件下，客户急切希望的某个性能目标不可能达到或是需要付出高昂的代价。如果相关的分析能尽早开展，客户就有可能在更早的时候做出决定，从而使这些目标成为可能。

如果客户期望的目标不能达成，他们会对最终交付的系统感到失望，哪怕系统其实足以满足业务需求。上述这些讨论有两方面的作用，不仅让开发团队了解客户的真实需求，而且让客户自

己也有一个清晰的目标。这样一来，只要系统达到了双方认可的目标，客户就会感到满意。有这些讨论作为基础，客户就不太会坚持不切实际的期望；如果他们仍然感到失望，至少那也是出于合理的原因。

何不让业务分析师一并采集这些需求

采集性能需求时不一定需要业务分析师在场，原因有几点。首先，此时功能需求的采集应该已经完成了；其次，即使业务分析师在场，开发者还是不能缺席，因为只有开发者才清楚分析性能问题需要获得哪些信息，也只有他们才能判断获得这些信息的途径和难度。性能测试人员应该提出前面介绍的这些问题，以此推动讨论进行，他们也能够判断每个需求是否容易测试。所以，当这些人坐在一起讨论时，业务分析师就可以把时间花在其他更有价值的地方。

小结

需求采集是为了让所有人都清楚最终交付的产品需要有怎样的性能才能支撑业务目标。之所以要让客户参与，是因为他们最了解自己的业务，这样才能确保采集到的需求足够准确。而且通过讨论也能帮助客户清晰自己对性能的需求，从而有效管理他们对系统的期望。

14.3　运行测试

下面我将简单讨论需要运行哪些测试以及何时运行它们。

运行哪些测试

所有频繁进行的用户操作都应该有对应的测试。这些测试应该记录吞吐量、错误率和响应时间的统计数据。然后你还应该复用这些测试，从而构建更复杂的测试。所有这些测试应该一起执行，尽可能地模拟真实情况，这样你就能从中获悉产品的性能状况。

准备好这些测试以后，就可以在不同的用户量、不同的数据规模下运行它们，观察性能数据的伸缩情况。如果可能的话，还应该在不同的机器数量下进行测试，从而了解增加硬件能给性能带来多大提升。从这几方面，你就能获知产品的伸缩能力。

最后，你还应该在超负荷的情况下进行测试，从而找出系统的失败点。还应该在用户分布情况基本不变的前提下加快用户操作速度进行测试。此外长时间运行性能测试有助于了解系统的可靠性。

何时运行测试

答案显然是越频繁越好。但显然这里有个问题，性能测试的本质决定了运行它们需要很长时间。尤其是可靠性测试，只有运行的时间足够长才有意义。所以不太可能为所有的构建都运行全套性能测试。我们既希望给开发者提供快速的反馈，又希望对系统进行全面的测试。

一个解决办法是找一台专门的机器，为最新的构建执行一组有限的性能测试。如果测试结果明显有别于前一构建，本构建就应该被视为失败。从中得到的结果虽不能代表系统的真实性能，但可以作为一个早期预警系统，让开发者们能很快知道自己的工作是否严重影响了产品的性能。

整套的性能测试应该尽可能频繁地在完整的性能环境下运行，可能每天运行几次。如果对环境的访问受到限制，在晚上执行性能测试也是个不错的折衷办法。

可靠性测试显然需要更长的时间，而且通常必须与其他性能测试运行在同样的环境下，也就是说没办法在工作日里进行。所以如果没办法找到一个专门用于可靠性测试的环境，在每个周末运行可靠性测试也是个解决办法。

在何处运行测试

如果可能的话，应该尽量让性能测试环境模拟真实的生产环境。如果生产环境太过庞大而无法整体模拟，那么就应该让性能测试环境模拟生产环境的一个部分，然后将真实的性能需求等比压缩到性能测试环境的水平。

如果无法得到专用的性能测试环境，事情就会变得比较棘手。如果必须和功能测试团队共享环境，那么可以考虑在夜里执行性能测试。此时，最好针对性能测试和功能测试使用不同的数据库，并用脚本来切换数据库，这样两组互不相容的测试就不会互相干扰。当然，这样做的前提是你能够在台式机上运行你的应用程序并编写性能测试。

在夜间运行测试需要注意，此时的网络情况常会与平时有所不同。网络可能不像白天那么繁忙，因为人们没有在工作；但也有可能数据备份或是别的批处理任务被放在晚上进行，因此占用大量网络流量。如果在性能测试的过程中突发大量网络活动，测试结果有可能受到明显的影响；如果性能测试和造成网络占用的任务恰好被计划在同一时间进行，测试结果就可能总是受到影响，以至于你看不出这种影响的存在，除非换个时间来运行测试。为此，应该安排在正常工作时间也运行一次性能测试，这样你才知道运行测试的时机是否是对结果造成明显影响的原因。如果不同时间运行测试的结果有很大差异，那么你可以尝试模拟正常工作时间的平均网络流量，或者

看这种结果差异是否保持一致，如果差异始终一致，就在查看结果时将其考虑在内即可。

根据我的经验，如果对测试环境存在明显的争用现象，就很有必要对其加以管理，谁在什么时候使用系统，使用哪个数据库，都应该事先规划好。有时测试环境下运行的测试会失败，在本地机器上却能通过，这时就需要把环境切换到性能测试的数据库，在问题修复之前其他QA都不能使用该环境。由于共享一个测试环境会限制运行性能测试的频度这一客观困难的存在，我们应该尽量尝试用一个单独的环境来运行性能测试，即便这个环境的配置与生产环境不很相似也没关系。

假如你面前摆着两个选择：一个测试环境与生产环境差异较大；另一个测试环境很接近生产环境，但只能在有限的时间（例如半夜）使用。你会怎么选？正确的答案是两个都要。你可以在独占的环境里编写性能测试，并且不受阻碍地频繁运行它们，这样你就可以把这组测试加入到持续集成系统中。接近生产配置的测试环境则是一个有价值的参考，你可以将其中得到的测试结果与从日常频繁运行的非生产系统中得到的测试结果相对比，从而了解这些测试结果与系统的最终性能究竟有何关系。

较小测试设备上的测试结果与生产环境的性能有何关系

一个常见的问题是测试环境的配置和生产环境不同。必须知道，如果测试环境和生产环境毫无相似之处，那么也就无法判断硬件对系统性能的影响。所以，如果不得不用一个较小的环境来做性能测试，应该怎么做？我的建议是模拟生产系统的有代表性的部分。下面我将介绍具体的做法。

以一个大容量web应用程序为例。系统的基本架构可能包括几台应用程序服务器、几台web服务器和几台数据库服务器。假设生产系统有N台数据库服务器（都是很高配置的机器）、2×N台应用程序服务器（配置较高）和4×N台web服务器（配置较差），那么你就可以考虑我的办法：准备一台数据库服务器，其性能大约是生产数据库服务器的一半；准备一台应用程序服务器，其配置和生产环境的应用程序服务器一样；再准备一台web服务器。

现在你就拥有了一个应用程序服务器与数据库服务器的组合，两者之间的相对性能比与生产环境一致，绝对性能值则大概降低了一半，并且web服务器的配置也不足。

在这个环境下，你可以直接访问应用程序服务器，借此了解应用程序的性能，从而掌握一台应用程序服务器所能达到的性能水平。然后你可以通过web服务器来访问，并且选择一种让web服务器成为瓶颈的测试场景，于是你就可以掌握一台web服务器所能达到的性能水平，而不会受到应用程序的影响。根据这两组数据，你应该能比较准确地判断在生产环境下各种服务器的配比

是否合适，并且对抱有一定程序信心的生产系统的性能有所预估。

需要记住的是由于每个web请求只能由一个应用程序服务器/数据库服务器的组合来处理，因此当服务器数量增加时，只有吞吐量的提升是能够确定的。而响应时间只会因为每台服务器的CPU计算能力或者内存增加而提升——假设系统负载水平保持不变的话。其原因在于，更快的CPU能以更快的速度处理更多请求，而更多的内存则让更多的信息得以缓存。

当然，以上讨论都基于一系列假设：机器配置始终保持不变，CPU都是同一家厂商制造的，操作系统都一样，数据库/web服务器/应用程序服务器的组合也保持不变。

请时刻牢记，测试环境与生产环境在机器配置、软件等方面差异越大，对真实系统的性能估计就越不准确。在前面的例子中，你可以把测试环境看作生产环境的一部分，从中估算出的生产环境性能就不会谬以千里。如果机器的配置与生产环境毫无可比性，用的软件也全然不同（例如把Oracle改成MySQL，把JBoss改成WebSphere），你仍然可以用这个环境来度量性能的变化情况，但根据测试结果估算出的生产环境性能数据就将非常可疑。

应该用多大规模的数据库来做性能测试

在做性能测试时必须记住，数据库的规模会显著影响从表中取出记录所需的时间。如果一张表没有合适的索引，当数据规模较小时可能还看不出问题；然而，一旦有几千行以上的生产数据，性能就会严重降低。

应该先与关注性能的客户交流，争取拿到一份生产数据库的副本，这样就可以针对它来进行测试。在这个过程中要注意数据保护，并对拿到的数据库做适当的清理，删除或修改其中的私密信息。

你还应该与客户探讨数据规模发生变化的可能性。数据量会大致保持在现有水平上吗？还是很可能会增长？如果会增长，增长的速度会有多快？只有了解这些信息，你才知道是否应该用一个比现在大得多的数据库来做性能测试。

要得到一个更大的数据库，最好的办法就是使用稳定性测试创建新的数据库。在稳定性测试中，你应该会创建新的用户和新的交易数据。如果这组测试整个周末都在顺利运行，那么你就能得到一个适用于未来情形的数据规模了。

如何处理第三方接口

如果系统用到很多第三方接口，性能测试最好不要直接去使用这些第三方系统。原因有两点：

首先，第三方系统可能并不适合成为性能测试的一部分；其次，即便第三方系统提供了测试环境，依赖你无法控制的第三方系统会降低测试的可靠性。

最好的办法是用一个单独的测试来获知第三方系统的平均响应时间，然后为它写一个mock或者stub，直接等待那么长的一段时间然后返回一个固定的响应。当然也可以直接返回响应，不过这样就会让测试失去了一些真实性，因为没有了等待第三方系统的时间，应用程序服务器就能够更快地释放数据库连接或者网络连接，这会给最终的测试结果造成差异。

需要多少种测试案例

这是个重要的问题，因为不恰当（过多或者过少）的测试量会严重歪曲测试结果。如果测试案例太少，所有相关的信息都会被缓存起来，系统就会显得比实际情况要快；如果测试案例太多，缓存就会溢出，系统就会显得比实际情况要慢。

多少种测试案例才是合适的呢？你需要和关注性能的客户讨论系统的预期使用情况，并且如果可能的话，请分析现有系统的使用日志从而找到一个答案。比如说，如果要测试的场景是“从应用程序获取顾客信息”，那么要在测试中覆盖到的顾客数显然和正常操作中涉及的顾客数有关。如果正常情况下系统中每天有5%的顾客信息记录会被取出，那么你的测试也就应该覆盖这么多的顾客。

为何要多次度量响应时间和吞吐量

一般来说，在从空闲状态开始增加负载量时，系统响应时间不会有什么变化。随着负载量不断上升，到达某一个点之后，尽管单位时间内处理的事务量仍然在上升（即吞吐量继续上升），但每个请求的响应时间会受到影响而逐渐上升。当服务器达到能力上限时，起初吞吐量会保持不变，而同时响应时间显著上升；最终吞吐量会急剧下跌，因为计算机已经无法承担所要求的如此大量的工作。这时响应时间将会飙升，整个系统会濒临死机。

在这个过程中，我们对几方面的信息感兴趣。首先，我们希望知道系统最大吞吐量出现在哪个点。除此之外，我们还希望知道最佳响应时间、当响应时间正好符合要求时的负载量以及负载达到事先测量的最大吞吐量的80%和90%时的响应时间等信息。

有了这组数据，你就可以限制每台应用程序服务器的连接数，从而确保系统性能始终保持在性能需求所规定的水平以上。可以看到，当负载量逼近极限时，响应时间会急剧上升；而当负载量达到80%甚至90%时，响应时间却没有太多变化。如果你必须确保某个性能水平，这一现象应该始终牢记于心。

有必要测试所有功能吗

对系统的所有功能进行测试基本上是不现实的。重点在于要覆盖最常用的功能。所以你需要识别出系统的主要使用场景，并针对这些场景创建不同的测试。

比如说，在线购物网站最主要的使用模式应该是“浏览”和“购买”。来购物的人不全会浏览很多页面，浏览的时间通常也不都会太长。所以你需要创建一个“浏览”的测试脚本和一个“购买”的测试脚本。为了让测试脚本更贴近真实，你需要知道用户浏览商品的平均数量、每次购买的平均商品数以及正常情况下一天时间内被浏览过的商品数占商品总数的百分比。

小结

关于性能测试，有很多话题可以讨论。需要度量什么？需要多频繁的测试？需要编写多少脚本？需要多少数据？在与关注性能的客户进行日常讨论时，主要应该确保这些重要的问题被提出来，并保证能得到你需要的信息。如果你认为事情的发展方式会严重影响测试的进展，也应该找时间与项目经理和客户沟通。

14.4　沟通

就测试的结果进行沟通是很重要的。所以接下来的问题就是，我们究竟在沟通些什么？所谓就测试结果进行沟通，可不止是报告几个数据那么简单。如果你这样做的话，团队里的所有人在身担其他任务的情况下都得花时间来分析这些结果数据。如果先对测试结果做一些基本的分析和解读，并给出一段概述，其他人理解起来就会容易得多。

所以你需要根据讨论出的性能需求和目前的性能水平来解读测试结果。首先，你需要指出系统性能与目标的接近程度（与目标的差距有多少，或是超出目标多少）。其次，你需要说明产品的性能是否发生了重大变化。不管这种变化是否导致产品无法达到性能目标，都应该将相关的信息告知所有相关人员。引起这种变化的原因可能是新增了一大块功能，这时对产品性能的影响是无可避免的，几乎没有能够改进的空间；但也可能是因为别的小问题，例如数据库缺少了某些索引，这样的问题应该很容易解决。

谁需要知道测试结果

有三组人需要了解测试结果：开发者、项目经理以及客户。开发者和项目经理应该在测试

运行完毕之后立即知道结果，这样他们就能在问题出现之初尽快合理地将问题解决。另一方面，没必要每天都拿一些小问题去打扰客户，否则当你说到真正重要的问题时他们就不会全神贯注；但也不应该疏忽与客户的交流。你可以每周安排一次会议，定期把性能测试的结果通报给客户。

此外，要记住不同的人会对不同的信息感兴趣。客户和项目经理可能希望看到比较高层面的概述，而开发者则对原始数据（以及给定时间内的响应数量等内容更感兴趣。如果能把适当的信息提供给不同的人，就能更有效地沟通产品的状态。

是否只需要写一份报告

不完全是。你当然可以写一份报告，然后用邮件发给所有人，但问题是大部分人很可能不会看这个报告，于是你想要传达的信息也就丢失了。报告只是一种用来帮助你传达信息的工具，而不是你的最终目的。

用一个网站向所有人展示最新的性能测试结果是很实用的。然后当你走到某个人的座位跟前讨论性能测试结果时，你就可以打开这个网站，把你发现的情况指给他看。不幸的是，大多数人并不善于看测试报告，所以为了确保你的信息能够传达到位，唯一的办法就是走到别人的座位旁边，或是拿起电话，向别人解释整个测试报告。

小结

你的目标是建立这样一种沟通机制：由于性能需求已经很清楚，因此无须拿着每次测试的结果去问客户是否可以接受；在每周介绍项目的当前性能状况时，你只需指出测试结果中的异常之处，并向客户解释异常情况出现的原因；如果某个区域的性能特别差，但经过判断这不是一个严重的问题，你就应该告诉客户为什么这块功能比较慢，为什么项目经理认为这不是一个高优先级的问题；如果客户不同意这个决定，那么项目经理和客户就应该坐下来具体讨论当前的情况。

14.5　流程

性能测试经常被放在项目结束前进行，这种安排严重影响了性能测试的效果。性能测试中最重要的事就是要定期地进行测试。如果直到项目最后几周才做性能测试，那么你将有很多事要干，而时间却非常紧迫。大部分时间会被用于编写测试脚本，并得到一些和产品有关的数据。这时你就会处于一种尴尬的境地，你大概知道系统运行得多快，但基本无从知道它是否足够快，而且也

没有时间做任何改进。

当第一段代码被编写出来，性能测试就应该开始了。虽然这时可能还没有任何可供测试的东西，但还是有很多事可以去做。你可以向开发者了解他们将要使用的技术，评估合适的工具，找出功能足以测试当前产品的工具。此外还需要识别出关注性能的客户，并且与他们一起启动需求采集的流程。

如何把各种工作连接起来

从这个阶段起，你的工作就开始进入一个循环。每周开始时，你会第一时间与关注性能的客户开会，讨论当前正在开发的功能的性能需求，同时介绍你的测试计划，以及如何展示需求得到了满足。客户也可以在这时要求更多的测试。剩下的几天里，你可以为最近完成的功能编写性能测试，维持已有的自动化测试，以及查看测试结果。一周将要结束时，你再次和关注性能的客户开会。这个会议有两项任务：首先是向客户展示本周编写的性能测试，并和客户讨论这些新的测试是否能表示产品满足早先提出的性能需求；其次是与客户一起查看现有性能测试的最新结果。

如何确保不拖后腿

只要按照这个每周的循环在工作，一旦性能测试的进度滞后，你很快就能清楚地看到。这时要想赶上进度，你可以增加用于性能测试的资源，也可以减少工作量。至于具体怎么做，很大程度上取决于性能需求对于项目有多重要。

你可以建立一个任务列表，将本周与客户所决定执行的测试任务都写在上面。然后你就可以与客户一起对这些测试排定优先级。每周你尽量完成列表上的任务。如果这样做下来导致测试覆盖率很成问题，那可能你需要投入更多人手来做性能测试；但也有可能在扔掉一些高难度、低优先级的测试之后，你完全能够保证足够的测试覆盖率，而又不会拖项目后腿。

如何确保每个问题都得到解决

必须在项目开始之初就和项目经理沟通，决定如何修复性能问题。你需要确保项目经理认同你的工作方式和你采集到的性能需求；你还要确保项目经理同意将性能问题作为bug提出，并且一旦性能问题出现就会有所行动，否则你就只能在项目结束时对着一大堆已知的性能问题徒呼奈何了。毕竟，如果性能问题出现之后不采取措施去解决，那么即使测出当前的性能水平也是毫无意义的。

14.6　总结

这个流程最大的好处在于它能确保你知道自己手上有什么、需要什么，而且你能肯定系统的每个部分都有测试覆盖，从而大大增加了发现问题解决问题的机会。让性能测试与开发同步，对每个功能都有测试覆盖，这样如果性能出了问题你就有时间应对。有一份性能需求在手上，你就能判断当前的系统是否需要改变。这份需求是客户根据业务流程和规模制订的，所以整个团队都对它有信心，大家也会乐于花时间来解决性能问题，因为他们知道这是一件有价值的工作。

参 考 书 目

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Reading, MA, first edition, 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring*: *Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

“内容非常精彩，本领域的必读之作。”

——DZone技术社区

“在帮助客户实施敏捷的过程中，ThoughtWorks人常被问到一个问题：有没有一套标准的‘敏捷模板’可供快速入门之用？作为一种强调持续改进的方法学，自然不会有一套放诸四海而皆准的‘标准流程’；但对于希望采用敏捷方法的组织和个人而言，若有一组普遍适用的最佳实践作为基础，便能少走许多弯路，以期事半功倍之效。本书正好满足了这一需要。”

——ThoughtWorks中国公司总经理，郭晓

The ThoughtWorks Anthology
软件开发沉思录
——ThoughtWorks文集

从编程技术到项目管理，Roy Singham、Martin Fowler、Rebecca Parsons等来自ThoughtWorks的思想领袖通过本书中的13篇美文，将自己多年沉思和实践所得倾囊相授，引领你走向敏捷软件开发的成功之路。

本书内容丰富，涵盖了软件开发的各个阶段，既包含DSL、SOA、多语言开发和领域驱动设计等热门主题，也有对象设计、一键发布、性能测试和项目管理等方面的经验之谈和独到见解。不论你是开发人员还是项目管理人员，都将从本书中获益匪浅。

作者简介

ThoughtWorks

ThoughtWorks公司于1993年在美国成立，现已经发展成为具有千人规模，在6个国家具有分公司的全球性IT咨询公司。公司汇集了许多业界思想领袖和众多高素质人才，致力于为客户解决最棘手最紧迫的问题，业务包括向客户交付定制应用软件、提供注重实效的咨询服务、协助企业进行敏捷开发以及开发软件等。

The Pragmatic Programmers

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议　计算机/网络技术/程序设计

人民邮电出版社网址：www.ptpress.com.cn

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn